前 言

各位好，我是何韬，一名“95后”程序员。在2021年冬天的一天，我在一个技术群中和朋友提到过我正在学习和整理单机文件系统的技术内容，希望能够梳理成文档。后来朋友提到为什么不考虑把这些资料系统性地整理一下，考虑出一本实体书呢？所以基于这样的建议，我在学习文件系统的时候，慢慢通过搜集和整理资料就有了这本书的雏形。当然，从写书到最终书稿内容的完善历时一年多，同时书中的内容章节也经过了多次的调整。我写完这本书稿时已经是2023年的春天，没想到反反复复地拖延了这么久。

我写这本书的一个目的是希望能够帮助一些对单机文件系统感兴趣的朋友进行深入学习。因为在和朋友交流的过程中发现单机文件系统的原理学习是分布式存储工程师的一个重难点，如果没有对底层文件系统的深入理解，那么在存储开发和运维过程中会碰到大量的参数需要优化调整，还有遇到生产故障等问题。

我在写这本书时也一直在思考，到底这本书要写成怎样的？市面上不乏Linux系统架构相关的经典书籍，但是专门讲解文件系统相关内容的，可以说很少，甚至是没有。大部分Linux书籍是对vfs的内容进行简单讲解浅尝辄止，也有一些资料是讲解如何实现一个简单的文件系统的，会在内核中注册一个自定义的文件系统，然后实现基础的创建目录操作等，然而这样的资料与分布式存储工程师的日常工作是相去甚远的，更准确地说，

这样的资料比较适合做一个类似文件系统的“hello world”的demo那样的小工具。因此在写这本书时，我希望能够结合自研单机存储引擎过程中遇到的一些困惑和感悟慢慢地分享出来。

读者对象

这本书的核心是希望能够让读者深入理解单机文件系统原理，因此对于本书的读者对象相对适合以下人群：

- 分布式存储工程师日常进阶学习文件系统原理。
- 对Linux文件系统感兴趣的朋友。

当然，Linux文件系统的深入学习相较于一些热门的技术方向，如前端和大数据等，会略显小众，因此本书希望读者具备Linux系统的操作基础知识，这是方便在学习的过程中，可以帮助理解对不同文件系统的使用命令。

如何阅读这本书

本书主要分为三大部分，其中第一部分是对Linux文件系统的宏观理解，主要是对文件结构和常见的文件操作语义的理解，以第一章为主要内容。第二部分则是以zfs为核心，深入理解zfs的部分模块实现和技术原理，以第二章为主要内容。第三部分则是以文件存储为核心出发，理解文件系统从单机到分布式过程的变化，还有文件系统的测试与优化等，以第三和第四章为主要内容。

我在写这本书时，已经在武汉成为一名分布式存储研发工程师，非常感谢武汉青云科技的同事，特别感谢团队同事黄蒙、宁安、肖文文、张文、黄力、杨俊、莫溢和任忠华，没有团队同事的协助和帮助，就不会有今天这本书。同时还非常感谢Zeppelin社区的刘勋、FastDFS的作者余庆和《elasticsearch源码解析与优化实战》的作者张超，以及黄亮和刘志旺先生，他们作为国内非常优秀的行业前辈，在我写本书的过程中给予了我很多的帮助和指导，十分感谢。另外还要感谢我的家里人，非常感谢他们的理解，写书的过程是一个很大胆且有挑战性的工作，他们曾经担心我写的书是否会有读者关注，能否卖得好，等等，这也让我对于内容和细节把控有更多的思考和想法（笔者邮箱：hiltontao96@163.com）。

推 荐 语

如果只是一般使用，选择文件存储可能有比较多的空间，包括各种商业和开源的。而对于存储软件产品的研发人员，或者需要自己运维开源存储的用户，想深入理解文件存储相关技术原理，研究源码的能力就比较重要了。

由 Ceph 引领的开源分布式存储热潮还没有过去，而 Ceph 并不是在各种应用场景下都是“万能”的，比如文件存储。我们看到历史悠久的 Lustre 仍然被 HPC 高性能计算行业追捧；功能特性丰富的单机文件系统 ZFS，早年以在 Solaris 和 FreeBSD 平台优秀的代码质量而著称，如今移植到 Linux 后应用也变得越来越多。许多对成本敏感的用户，在分布式 Lustre 的后端，用 ZFS 替代相对昂贵的传统商业 SAN 存储。

在开源项目流行的时代，经验和技巧的分享是很有价值，对整个行业都是有益的。我很高兴看到一名存储同行、“95 后”程序员何韬撰写出新书。

从本书的目录章节“初始 Linux 文件系统”——“深入理解 zfs”——“文件存储从单机到分布式”——“测试与优化”，并最后落到“lustre 对 zfs 参数优化”，是很贴近实用的，希望这本书能够帮助到更多的技术同行朋友！

《企业存储技术》微信公众号作者　黄亮

2023 年春天

目 录

contents

1、初始Linux文件系统

1.1 为什么学习文件系统

在深入学习了解文件系统之前，我们需要先了解一下为什么需要学习文件系统？如果要学习文件系统，那么该学习哪些内容呢？学了文件系统之后又有什么作用呢？对于这些常见的疑惑，最好的答案就是我们需要先了解一下分布式存储工程师的岗位要求。

如果各位时不时去关注一些招聘网站中对于分布式存储岗位的技能要求，以下四点往往出现的频率很高。

- 熟悉 scsi/nvme/nfs/smb 等存储协议
- 熟悉 rocksdb/rmda/spdk/dpdk 等技术
- 熟悉 glusterfs/lustre/ceph 等分布式存储系统
- 熟悉文件存储高级特性实现原理，如快照、QoS 和多租户等

对于以上这些技能要求，虽然没有提到文件系统，但是对于存储协议的原理，是基于文件系统的文件结构和语义的，同时 lustre 和 glusterfs 这些主流的分布式存储系统，也是需要依赖单机文件系统，并且都会对文件系统，如 zfs 有一些参数优化设置，因此，如果没有深入了解过单机文件系统原理，往往有很多参数的设置和实现很难把握住。

除此之外，目前越来越多优秀的存储厂商开始考虑自研的

单机存储引擎，需要适配上层的分布式系统，如 bluestore 和 filestore 等。因此深入学习文件系统的原理实现，可以看懂上层技术的本质，也可以更加好地把握住技术实现变更优化方向，掌握技术核心竞争力。

当然在学习分布式存储技术的时候，往往会被问到是否需要专门学习 c/c++ 语言，尤其是对于 c++ 语言，这门语言的技术学习门槛相对较高，语言特性也比较复杂，有时候投入了大量的精力学习了 c++ 语言的技术原理，但是似乎又感觉和技术岗位不太匹配。

对于这样的困惑，笔者目前不建议去学习 c++ 语言，因为语言是要为项目服务的，如果单纯去看语言，是无法理解到底需要学到什么程度的，同时也很难结合项目去看，笔者更加建议看到项目不懂的地方，再去查相关的语法，哪里不懂学哪里，这样效率可能会更高。当然，笔者目前日常使用 Rust，这个语言如果各位没有需求，也不建议马上深入学习，经常会入门反反复复，学了却似懂非懂，不知道到底该怎么日常使用了。

那么接着我们就需要知道文件系统中该学习什么模块的内容，又该从哪里开始学习呢？对于文件系统，从日常接触中可以感知到的就是对文件的操作，文件结构是一个重点，因为所有的操作命令最终都是要对文件进行操作的，也就是常说的增删改查。对于增删改查，其实并不只是我们常见到的简单情况，例如文件的写入也是区分不同情况的，包括覆盖写、对齐写、非对齐写和追加写等场景，对于这些不同的场景，就是文件结构操作的实现细节差异了。对于这部分内容，在后面的小节中会分享。

除了文件结构，文件系统最终也是要把数据持久化到磁盘中的，通常是机械硬盘或者固态，而写入数据的机制，则是被称为数据事务落盘（transaction group）；除此之外，还有文件系统的空间管理等模块，对于这些内容，本书会以 zfs 为核心进行深入分享讲解，在后面的章节中进行分享。

1.2 文件语义

很多时候，提到 Linux 的时候，尤其是文件系统，那么必然绕不开 posix 标准这个概念，对于这个概念的解释，网上有很多，但是知道了这些内容以后有什么作用呢？研发人员需要知道 posix 语义吗？如果是做分布式存储，尤其是研究单机文件系统的研发，还需要深入研究吗？带着这些问题，我们一点点来探究其中的奥秘。

简单点来说，posix 就是一套标准规范，是为了跨平台使用的，而平时要是使用了 glibc 中的 api，或者调用了 Linux 的系统调用，如 write，open 等，那么这些函数需要有一个标准，这个就是 posix 标准了。如果再粗糙点解释，有点类似今天大家所熟悉的 Linux 的 RestFul 标准（其实这样比喻是非常不恰当的，不过可以帮助了解）。

当然，对于这个标准有很多函数，这些函数为什么需要考虑呢？因为文件系统中，系统调用接口也需要遵循这样的标准。接触系统调用，通常都是从 VFS 开始的，因此先简单了解一下 Linux 中的文件系统架构，如 1–1 所示。

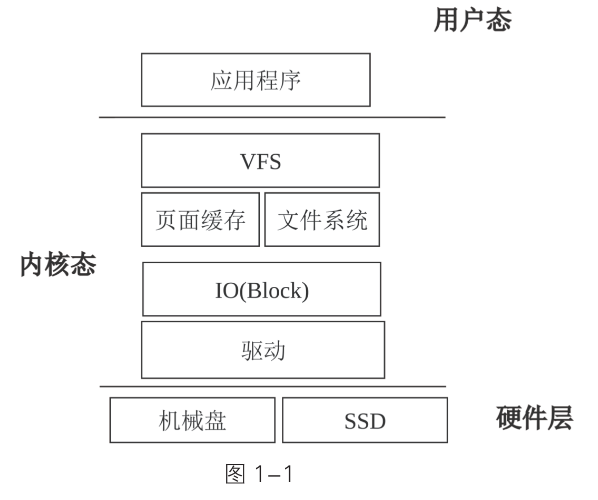


图 1–1

VFS 可以说就是一个入口，不管是 xfs，zfs 还是 ext4 等文件系统，在执行的时候，都要遵循 VFS 上的系统调用接口。

既然 VFS 是入口，那么常见的系统调用，一般会涉及哪些呢？大家通常会想到和接触比较多的就是增删改查，也就是 read，write，open 这些，但是对于删除，这里有一个语义 truncate，可能大家接触不多（数据库中也有这个命令，但是使用场景和文件系统不同，数据库中通常是用于删除数据的）。那么在日常研发中，该如何知道这些语义标准呢？最简单的办法就是在 Linux 上查看，如使用命令

```
1. ~# man 2 truncate
2. ...
3. If the file previously was larger than this size, the extra
   data is lost. If the file previously was shorter, it is exten
   ded, and the extended part reads as null bytes ('\0').
```

代码 1–1

从上面的描述内容可以看到，truncate 是可以让文件变大变小的，如果让文件变小了之后，可能会丢失掉文件数据，也就是如图 1–2 所示。

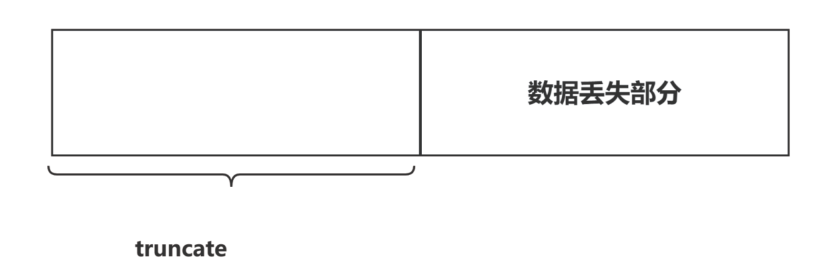


图 1–2

那么什么时候需要使用到 truncate 这样的命令呢？有一种场景叫覆盖写：需要先把文件的数据全部删掉，然后只写入部分数据，并不一定会和原来写入的大小相同（可能会多，也可能会少）。这时候需要先把文件 truncate 为 0，表示把文件原来所有的数据删掉，接着再写入。

对于数据写入，这里还要留意一个问题，如果写入的时候，对于某个数据块不是全覆盖写的话，若只写一半，例如 4KB 大小的数据块，但是实际数据只是写入了前面的 2KB，如图 1–3 所示。

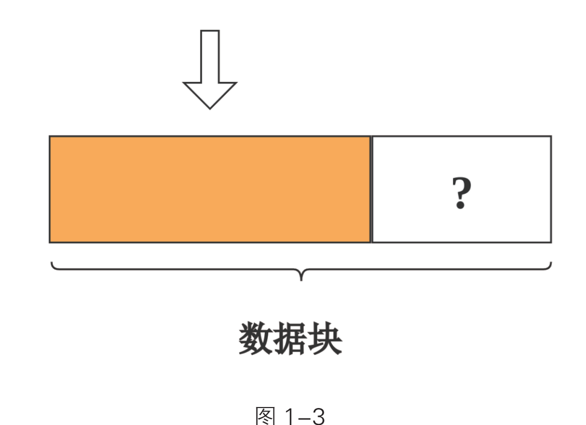


图 1–3

这里就无法知道后面这部分内容到底是什么了。这里造成的后果就是，可能会在计算文件校验码的时候，各个节点的校验结果会不同（因为无法保证数据块一定是完全相同的，主要就是没被覆盖写的部分），所以在使用数据块的时候，要保证数据初始化。同时请留意，有时候部分写，也会称为非对齐写，因为并不是对整个数据块进行覆盖写入，与此相对则是对齐写。关于这部分，后面的小节会深入分享一下。

写入的时候，一般是顺序写，或者追加写，也就是 write 从头开始写入，又或者像日志文件那样，从文件末尾开始追加写。但是总有一些奇怪的情况出现，例如能否只写中间一部分呢？甚至出现写一部分，然后空着跳跃，再写文件的另一部分，这就是空洞文件的来源之一（这里当然还有其他场景会触发空洞文件，例如一边写入，一边被修复），如图 1–4 所示。

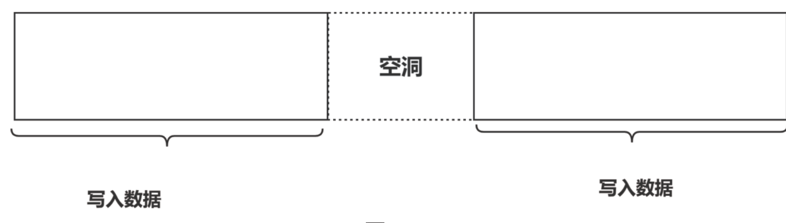


图 1–4

想要实现这个效果，这时候系统调用 lseek 就派上用场了。使用命令查阅 lseek 命令文档的时候，大家会发现 Linux 文档中会提示各位要非常小心文件的空洞，有时候应用程序是无法发现或者理解的。对于文件空洞，是填充数据 0，还是没有写入数据呢？这也是有区别的。数字 0 也是一个数值，而空洞则是指没有数据，因此要留意二者的区别。

日常中一些常见的中间数据写入的类比场景，可以用 WPS 文档来举例，平时写文档的时候，第一次可能是从前往后写入，但是往往需要修改或者润色部分内容，因此对于中间的页面内

容会有修改的情况。

那么下面看看一些常见的使用 truncate 的场景。

```
1.  //终端1
2.  ~# tail -f /tmp/1.txt
3.
4.  Sun Aug  7 13:21:18 CST 2022
5.  Sun Aug  7 13:21:19 CST 2022
6.  Sun Aug  7 13:21:20 CST 2022
7.  Sun Aug  7 13:21:23 CST 2022
8.  Sun Aug  7 13:21:24 CST 2022
9.  tail: /tmp/1.txt: file truncated
10.
11.
12.
13. //终端2
14. ~# date >> /tmp/1.txt
15. ....
16. ~# echo > /tmp/1.txt
```

代码 1–2

这里在两个终端中进行操作，其中一个使用 tail 命令不断监视查看文件输出内容，另外一个终端输入一些数据后，使用 echo 命令把日志内容清空，这里就可以看到在 tail 命令的终端里，文件显示已经被 truncate 了。对于 truncate 语义的内容，后面小节也会专门分享一下具体的实现内容，因为 truncate 的实现，往往会和写入有关，有部分内容是平时可能不太容易关注到的。

那么简单了解了系统调用之后，也算是对 VFS 中的作用有一些简单的接触了。对于一个正常的应用服务，如果要读写本地磁盘数据，会向本地的文件系统发起请求，例如使用 mkdir 或者 touch 这样的 shell 命令进行操作，那么首先会经过 VFS 层，这个其实是 linux 操作系统中对于文件系统的一个规范要求，也就是说，不管是自定义的操作系统，还是一些 ext4，都需要遵守一些规范，而 VFS 中最核心的元素就是 inode，dentry，

superblock 和 file 四个元素。

inode 是负责记录每个文件的一些元数据信息的（在 linux 系统中，一切皆文件，不管是目录还是 devices 设备，都会封装为一个文件一样调用）。dentry 对象则是用于记录文件的结构关系的，也就是不同目录的上下级层级结构关系和树状关系等。superblock 就是超级块，用于管理整个操作系统中 inode 资源整体使用情况等，如果把文件系统比作一个图书馆，那么图书馆里面的每一本书就是 block，而书的分类和标签信息就是 inode，superblock 就是统计整理整个图书馆的资源情况的。file 对象就是用于记录每一本书的租借情况，对于文件来说，这个文件是否被打开过，是否产生了文件句柄 fd（所谓的文件句柄可以理解为，在 file 对象和打开的进程的数据结构中做了一些标记）等。

1.3 文件结构

认识了 VFS 和一些有趣的系统调用后，不管任何的系统调用，在文件系统中，最终都是为了对文件进行操作的，因此文件到底是怎样的？文件的结构又有哪些内容？这些东西往往是一个文件系统的核心，因为对于文件读写和数据落地等，都是需要基于文件结构出发来设计的，因此下面先简单地来了解一下何为文件。

可能大部分人对于文件的印象如图 1–5 所示，脑海中文件就是一个长方形的图片，里面写满了数据，一般调用读写请求时，关心的是 offset 和 count 参数，也就是读写文件的位置和长度，决定了文件从哪里开始读写，并且读写多长的数据，还要关心的是文件的大小 size 等信息。

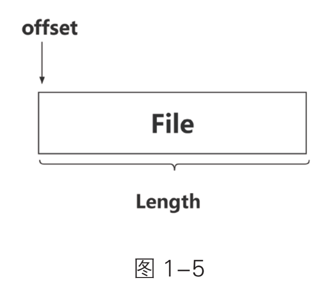


图 1–5

但是实际上文件的结构远远并不是如此简单的，因为文件上面还有一些其他的元信息（元信息的概念对应不同层次结构意义是不同的，分布式系统中的集群也有元信息，文件有元信息，请注意区分不同），下面使用 stat 命令展示一下。

```
1.  ~# stat 1.txt
2.    File: 1.txt
3.    Size: 0      Blocks: 0     IO Block: 4096    regular empty file
4.  Device: fc01h/64513d    Inode: 1543          Links: 1
5.  Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
6.  Access: 2022-06-29 00:38:24.722798707 +0800
7.  Modify: 2022-06-29 00:38:24.722798707 +0800
8.  Change: 2022-06-29 00:38:24.722798707 +0800
9.   Birth: -
```

代码 1–3

从上面的 stat 命令可以看到，对于文件 1.txt，这里还记录了三个时间，分别是 access time，modify time 和 change time，还有文件的 id，也就是 Inode Id，这些信息是每个 Linux 文件中都会具备的，如果文件只是单纯记录了数据，那上面 stat 命令显示的元数据显然是没地方存放的，因此文件结构并不是如代码 1–3 所示如此简单，这里肯定还有地方是专门存储上述文件 stat 信息的。

文件除了记录上面提到的信息，还要记录数据，要记录的东西很多，而且长度不定（对于元数据信息，也是可以设置的，使用 setxattr 等命令）。当然这里有部分信息是固定的、必需的，如上面 stat 中展示的信息，那么这里就可以把文件中的信息分为元数据和数据两种，并且元数据中也有固定和非固定的。

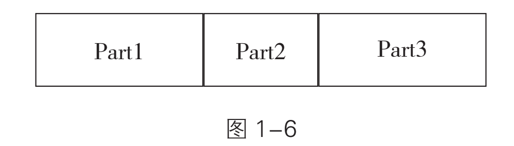


图 1–6

这时候可以考虑把文件分为三部分了，如图 1–6 所示，其中 part1 是存放一些固定的文件元数据信息的，part2 就是存放用户自定义的信息的，part3 则是存放文件数据的。对于 stat 中要展示的固定元数据信息，这里的长度都是确定的，因此 part1 的长度是可以固定下来的。

但是对于 part2 和 part3，这里是无法确定大小的（因为无法知道用户写多少数据，或者写入多少扩展属性），因此文件可能随时在变化，该如何处理呢？文件的大小没有上限（只要单机文件系统能够支持，并且不超过磁盘上限），但一个文件上百 G 甚至上 TB 也是正常的。那么这里对于 part 3 来说，这里的数据是可以不断追加大小的，但是考虑这里的文件结构看起来是长方形的，那么在实际的单机文件系统中，文件是否就是类似这样的结构，也就是一个数组呢？答案当然不是，大家还记得讲系统调用的时候提到的 truncate 和 lseek 命令吗？这里是可以删掉部分空间的，甚至可以做到写一个空洞文件出来，如果使用数组，那么该如何表示空洞文件呢？这里就引出了另外一个数据结构——树。

常见的数据结构中，树也是其中一个，二叉树则是大学学习数据结构时必不可少的一个。那么这里在设计 part 3 的时候，

能否对存放数据的部分使用二叉树呢？也就是如图 1–7 所示。

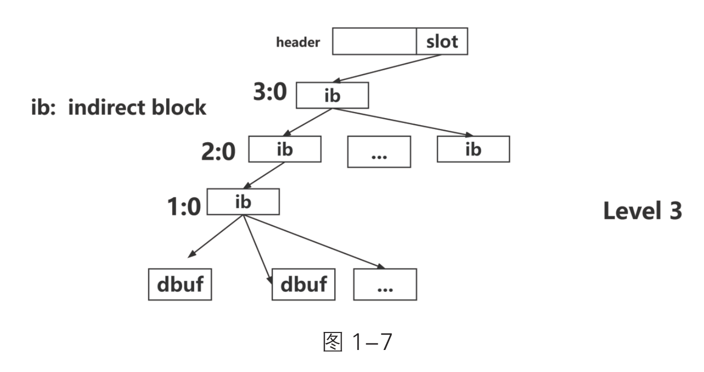


图 1–7

这里出现了一种新的文件结构示意图，对于这种树形的文件结构，在文件系统中是非常常见的（xfs 使用 b+ tree，zfs 中使用的是 avl tree，对于这些差异，暂时可以不用理会，不影响对主体结构的理解）。那么使用了这样的结构之后，有什么好处呢？

在讲解文件的树形结构之前，需要讲一下文件的数据其实并不一定会连续存放，因为当数据较多时，哪怕是一次性写入，在缓存中累积然后一次性申请很大的连续空间写入，虽然这样做对读取数据和磁盘写入有提高，但是在实际的工程应用中，数据缓存累积过多或者累积时间过长，会造成应用的超时。如果只是写入缓存就以为成功，那么一旦机器宕机了，缓存中的数据是会丢失的，对于存储系统来说是最不可接受的事情。同时哪怕是在内存中缓存了很多数据，实际写入的时候，早期的磁盘写入性能都是很慢的，也是需要一部分一部分数据慢慢写入的，只是把数据弄成了顺序写入，那样会提高一些效率（相对的是随机写入，如同一文件写入数据，这次可能在磁盘扇区的一边，下一次在另外一边，又或者是频繁更换柱面来写入数据，对于机械盘来说，这种数据写入方式效率是很差的）。

因此出于多种原因考虑，不管是系统使用还是机械物理特

性，对文件写入的数据都需要进行切分，这时候切分的大小一般就叫作 block，通常常见的大小有 4k、8k、32k 和 64k 等。对于一些分布式存储系统，可能会以 M 为单位，但是实际上在写入单机节点的时候，数据落盘还是会根据文件系统的 block 大小来确定的。如无特殊说明，一般默认说 block size 是指 4k，这是最常见的。

对于一个文件，头部是固定大小的，其中包括了前面提到的 part 1 的文件元数据信息等，还有一些预留字段和锁字段，同时还有一些字段，是为了知道当前文件的树形结构特性的（如文件的 level 层级）。像图 1–7 中，这里叶子节点是真实的文件数据，而 ib 都是被称为间接块（indirect block，简称 ib），当文件很大的时候，就只是需要扩展一下树形结构即可，文件的大小可以有更多的变化和层级。注意 indirect block 和 dbuf 的称呼是 zfs 中的，其他文件系统也有类似的说法，大家了解即可。

同时这里规定了每一个 ib 块可指向的子节点数量是有限的（这样方便申请空间的时候使用，同时计算树的层级是方便的）。那么就意味着，平常说的改变 block size，就是在改变数据 block，就是 dbuf。

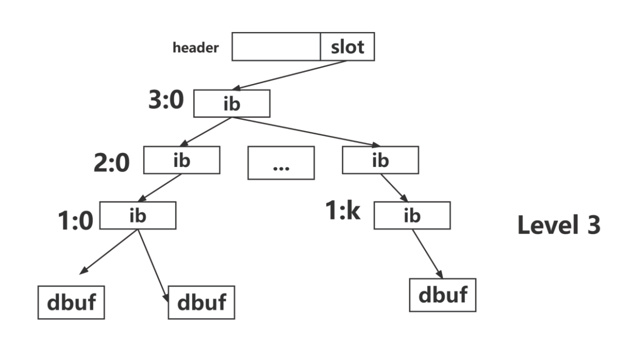


图 1–8

那么这里再结合一下前面提到的空洞文件，其实就是最底层的叶子节点中，并不是写入每个 ib 对应的子节点，甚至可能会出现部分 1∶0 层级中的 ib 是不存在的。

同时这里如果要计算某个数据块 dbuf 的位置，又该如何计算呢？因为 ib 中可以指向的子节点是固定的，假设为 n，那么如图 1–8 中 1∶k 的 ib 所指向的 dbuf，假设为该 ib 的 0:（n–1）的子块，那么该 dbuf 在整个树形结构中的位置就是 k*n+（n–1）。这里就有点类似相对路径和绝对路径的区别。

当然还可能会出现一些小文件，有多小呢？数据可能并没有达到一个 block 大小，甚至只有几字节的大小，如果这时候使用树形结构，那岂不是非常浪费空间。因此文件系统中，为了优化这些内容，在文件结构里面，会把数据放到 header 头部中，这种结构在 xfs 中叫作 short format。除此之外还有一些特殊的结构，如 zfs 中有 gang block 和 spill block，这些遇到后再查阅资料即可。图 1–9 则是一个 level 0 的结构示意图。

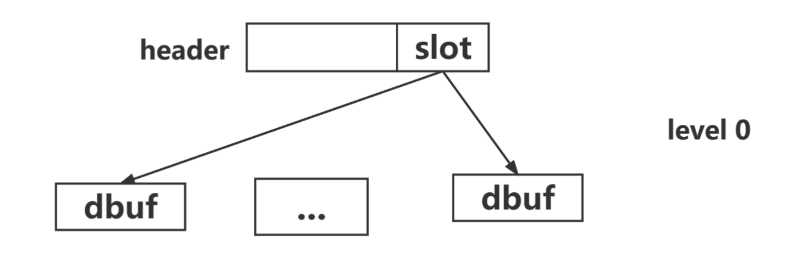


图 1–9

在文件系统中，为了优化文件结构，考虑设计出很多方案。当文件的数据不大不小时，也就是文件数据的大小并没有小到可以放进文件 header 结构里面，同时也并没有很大，可以组成多层的文件 level 结构，这时候又同时恰好文件 header 中预留的 slot 可以指向这些实际文件数据（slot 可以理解为一个指针结构，用于保留指向文件数据块或者间接块的），这时候就有了图 1–9

所示的这种文件结构了，在 xfs 中，这被称为 XFS_DINODE_FMT_EXTENTS，也就是 extend 结构。那么这种文件结构，其实就是树形结构的最开始状态，也就是 level 0 的状态，和其他 level 层级结构不同，这里文件的 header 中的 slot 是指向了数据块 dbuf 的，也就是没有了 ib 块。

```
1.  //这是xfs的文件结构，在fs/xfs/libs/xfs_format.h中
2.  typedef struct xfs_dinode {
3.            ...
4.             __u8 di_format;
5.            __be64 di_nblocks;
6.           __u8  di_forkoff;
7.  }
8.  enum xfs_dinode_fmt {
9.   XFS_DINODE_FMT_DEV,   /* xfs_dev_t */
10.  XFS_DINODE_FMT_LOCAL,  /* bulk data */
11.  XFS_DINODE_FMT_EXTENTS,  /* struct xfs_bmbt_rec */
12.  XFS_DINODE_FMT_BTREE,  /* struct xfs_bmdr_block */
13.  XFS_DINODE_FMT_UUID   /* added long ago, but never used */
14. };
```

代码 1–4

来看一下 xfs 文件结构中的一些字段，为了方便查看，可以在 linux repo[2] 下载。这里 xfs 中提到的三种文件结构 LOCAL，EXTENTS 和 BTREE 就是上述提到的三种情况，LOCAL 就是数据内嵌到了文件结构内部，也叫作 short format；EXTENTS 对应 level 0 的情况，而 BTREE 则是对应 level 大于 0 的情况。

在 xfs_dinode 的结构体中，有一个字段叫 di_forkoff，这个字段的作用就是用来区分 part 2 和 part 3 的间隙的，如图 1–10 所示，因为在 xfs 的设计中（按照目前最新的 5.x 设计），虽然整个 dinode 结构体大小是确定的，然而对于 part 2 和 part 3 之间的大小是动态的，那么 forkoff 这个字段就是用来分隔这两部分的界限了。



图 1–10

对于 part 2 和 part 3 的名称，在 xfs 中分别称为 data fork 和 attribute fork.

另外在 ext4 和 xfs 文件系统中，extend 的概念可以理解为底层是连续的 block 数组，而 zfs 中则没有使用该概念，还是使用 block，一般常见的 block size，还是 4k、8k、16k，对于这个的差异，目前来说并没有太大的影响，大家了解即可。

学习了前面的知识之后，大家可以思考一下，如果 block 的大小改为了 32k，那么这里是改变数据块的大小，还是会改变间接块的大小呢?

1.4 file 和 inode operation

对于 VFS，了解过的朋友对 inode，file 和 dentry 这三个结构不太陌生，那么对于一个文件系统来说，在 Linux 中结构是 struct file 结构，如下所示。

对于 Linux 的源码，如无特殊说明，默认 Linux 5.x。

```
1. struct inode {
2.   const struct inode_operations *i_op;
3.
4.   union {
```

```
5.     const struct file_operations *i_fop; /* former ->i_op->default_file_ops */
6.    void (*free_inode)(struct inode *);
7.     };
8.     ...
9. }
```

代码 1–5

对于 inode 结构体，这里非常值得关注的字段，分别就是 inode_operations 和 file_operations 字段，顾名思义，operation 翻译过来是操作，那么上述两个字段分别就是对 inode 和 file 的操作，因此，这里有什么操作呢？为什么需要拆分为两个操作呢？为了理解这些问题，需要进一步看看这两个字段的具体内容。

```
1. struct inode_operations {
2.  struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
3.
4.  int (*create) (struct user_namespace *, struct inode *, struct dentry *,
5.          umode_t, bool);
6.
7.  int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *,
8.         umode_t);
9.  int (*rmdir) (struct inode *,struct dentry *);
10.  int (*mknod) (struct user_namespace *, struct inode *, struct dentry *,
11.         umode_t,dev_t);
12.  int (*rename) (struct user_namespace *, struct inode *, struct dentry *,
13.    struct inode *, struct dentry *, unsigned int);
14.  int (*setattr) (struct user_namespace *, struct dentry *,
15.    struct iattr *);
16.  int (*getattr) (struct user_namespace *, const struct path *,
17.    struct kstat *, u32, unsigned int);
18.        ...
19. }
```

代码 1–6

这里看到 inode_operation 中，主要就是创建修改和设置 inode 的扩展属性，另外这里还有一个叫 lookup 的，在 Linux 中，因为目录结构是树形结构，因此想要查找到一个文件，必须知道该文件的 inode 结构，这样才能准确定位到文件的数据位置和其他信息，因此 lookup 就是做这个功能的。

现在假设要读取一个路径为 /a/b/ 下的 c 文件，那么这里按照绝对路径，则是 /a/b/c，每次需要先获取到上级目录，然后才能定位到该目录下的子目录或者文件。查找的时候，这里会对目录文件的名称作 hash 的，这样便于做索引优化查询速度。

而在 Linux 中，关于查询的内容，可以见 Linux 内核代码 fs/namei.c 中的 link_path_walk 函数。在该函数中有一段无限循环代码，如下所示。

```
1. static int link_path_walk(const char *name, struct nameidata *nd)
2. {
3.     ...
4.     for(;;) {
5.         ...
6.         if (unlikely(!*name)) {
7.             link = walk_component(nd, 0);
8.         }else{
9.             link = walk_component(nd, WALK_MORE);
10.         }
11.     }
12. }
```

代码 1–7

这里的无限循环，就是不断遍历解析路径的，解析的时候，会遇到各种情况，例如当前目录或者文件是一个连接，那么需要调用其他函数来处理的，又或者该目录是一个挂载点，那么需要跳转到对应的文件系统下来进行处理。正常来说，对于遍历，这里会调用 walk_component，接着会调用 lookup_fast，那么

这里就会涉及 __d_lookup_rcu 函数来最终寻找到文件的 inode 信息，该函数会和 rcu 的内容有关。

RCU 其实是内核中的一个机制，对于读多写少的情况何为多和少呢？根据 Linux 下的文档说明，如果一个结构，写入的次数大于读取次数的 10%，那么写就不算少了，当然这里还要结合具体情况来看，而对于这样的结构，例如链表或者数组的话，是可以做一些优化的，主要就是拆分链表减少加锁带来的时间消耗，关于这部分内容，可以具体看 linux 源码的文档，在 Documentation/RCU 下。

```
1.  struct file_operations {
2.   struct module *owner;
3.    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
4.    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
5.   ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
6.   ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
7.   int (*mmap) (struct file *, struct vm_area_struct *);
8.   unsigned long mmap_supported_flags;
9.   int (*open) (struct inode *, struct file *);
10.  int (*flush) (struct file *, fl_owner_t id);
11.  int (*release) (struct inode *, struct file *);
12.  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
13.  int (*fasync) (int, struct file *, int);
14.  int (*lock) (struct file *, int, struct file_lock *);
15.          ......
16. }
```

代码 1-8

从 inode_operations 和 file_operations 的内容来说，其实就是对应着文件结构中，对文件元信息的操作和数据的操作。

这里讲了这些内容以后，可以来思考一个问题，为什么 Linux 不允许创建一个同名的目录和文件呢？


```
1. # touch 1
2. # mkdir 1
3. mkdir: cannot create directory ‘1’: File exists
```

代码 1-9

出现这个问题的原因，和前面提到的查询过程有关，因为同名的不管是目录还是文件，都会被称作 hash，那么会得到一个相同的 hash 结果出来，无法通过校验。同时因为像 open 的时候，规定的系统调用函数并没有给出参数来指明当前的路径名称是一个什么类型的文件，因为有可能是链接，设备等，无法区分不同类型的同名文件。

```
1. int open(const char *pathname, int flags, mode_t mode);
```

代码 1-10

1.5 注册与卸载文件系统

如果想要自己注册一个文件系统，那么可以参考 Linux 中的源码，例如参考一下 xfs 中的内容，首先有一个 file_system_type 字段，这里定义了该文件系统的名称类型等信息。

```
1. static struct file_system_type xfs_fs_type = {
2.   .owner    = THIS_MODULE,
3.   .name    = "xfs",
4.   .init_fs_context = xfs_init_fs_context,
5.   .parameters   = xfs_fs_parameters,
6.   .kill_sb   = kill_block_super,
7.   .fs_flags   = FS_REQUIRES_DEV | FS_ALLOW_IDMAP,
8. };
```

代码 1-11

这里可以看到定义的文件系统名称为 xfs，然后有 xfs_fs_type 变量之后，需要注册到内核中，这里的函数是 register_

filesystem。

```
1.  STATIC int __init
2.  init_xfs_fs(void)
3.  {
4.      ...
5.      xfs_dir_startup();
6.      error = xfs_cpu_hotplug_init();
7.      ...
8.      error = register_filesystem(&xfs_fs_type);
9.
10. }
```

代码 1–12

当然，有注册就该有卸载，卸载的函数名称叫作 unregister_filesystem，如代码 1–13 所示：

```
1.  STATIC void __exit
2.  exit_xfs_fs(void)
3.  {
4.   xfs_qm_exit();
5.   unregister_filesystem(&xfs_fs_type);
6.       ...
7.   xfs_mru_cache_uninit();
8.   xfs_destroy_workqueues();
9.   xfs_destroy_zones();
10.  xfs_uuid_table_free();
11.  xfs_cpu_hotplug_destroy();
12. }
```

代码 1–13

因此如果想要了解一个文件系统的启动和卸载，都可以考虑从这两个函数入手。当然除了这些，还要定义非常多的操作函数，这些就是与 file operation 相关的信息了，前面有提到过。感兴趣的朋友可以沿着这样的路线来对源码展开学习，这也是一个文件系统学习的切入点。

1.6 接口错误码

在讲解常见的错误码之前，需要先分享两个常见的系统调用命令，主要用于辅助查看错误码信息的。

1.6.1strace 命令

这个命令主要的用途就是查看系统命令的调用，如下所示。

```
1.  $sudo strace touch /tmp/abc/1.txt
2.  execve("/usr/bin/touch", ["touch", "/tmp/abc/1.txt"], 0x7ffe71079ae8 /* 17 vars */) = 0
3.  brk(NULL)                               = 0x12ee000
4.  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
5.  openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6.  fstat(3, {st_mode=S_IFREG|0644, st_size=139933, ...}) = 0
7.  mmap(NULL, 139933, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9b5f7fb000
8.  close(3)                                = 0
9.  openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
10. read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260A\2\0\0\0\0"..., 832) = 832
11. fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
12. mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9b5f7f9000
13. mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9b5f638000
14. mprotect(0x7f9b5f65a000, 1658880, PROT_NONE) = 0
15. mmap(0x7f9b5f65a000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f9b5f65a000
16. mmap(0x7f9b5f7a2000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7f9b5f7a2000
17. mmap(0x7f9b5f7ef000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000) = 0x7f9b5f7ef000
18. mmap(0x7f9b5f7f5000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f9b5f7f5000
19. close(3)                                = 0
20. arch_prctl(ARCH_SET_FS, 0x7f9b5f7fa580) = 0
21. mprotect(0x7f9b5f7ef000, 16384, PROT_READ) = 0
```

```
22. mprotect(0x416000, 4096, PROT_READ)      = 0
23. mprotect(0x7f9b5f845000, 4096, PROT_READ) = 0
24. munmap(0x7f9b5f7fb000, 139933)           = 0
25. brk(NULL)                                = 0x12ee000
26. brk(0x130f000)                           = 0x130f000
27. openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_
    RDONLY|O_CLOEXEC) = 3
28. fstat(3, {st_mode=S_IFREG|0644, st_size=10988896, ...}) = 0
29. mmap(NULL, 10988896, PROT_READ, MAP_PRIVATE, 3, 0)
    = 0x7f9b5ebbd000
30. close(3)                                 = 0
31. openat(AT_FDCWD, "/tmp/abc/1.txt", O_WRONLY|O_CREAT|O_
    NOCTTY|O_NONBLOCK, 0666) = -1 ENOENT (No such file or direc
    tory)
32. utimensat(AT_FDCWD, "/tmp/abc/1.txt", NULL, 0) = -1 ENOENT (N
    o such file or directory)
33. openat(AT_FDCWD, "/usr/share/locale/locale.alias", O_
    RDONLY|O_CLOEXEC) = 3
34. fstat(3, {st_mode=S_IFREG|0644, st_size=2995, ...}) = 0
35. read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
36. read(3, "", 4096)                        = 0
37. close(3)                                 = 0
38. openat(AT_FDCWD, "/usr/share/locale/en_GB/LC_MESSAGES/
    coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or direc
    tory)
39. openat(AT_FDCWD, "/usr/share/locale/en/LC_MESSAGES/coreutils.
    mo", O_RDONLY) = -1 ENOENT (No such file or directory)
40. write(2, "touch: ", 7touch: )                  = 7
41. write(2, "cannot touch '/tmp/abc/1.txt'", 29cannot touch '/
    tmp/abc/1.txt') = 29
42. openat(AT_FDCWD, "/usr/share/locale/en_GB/LC_MESSAGES/libc.
    mo", O_RDONLY) = 3
43. fstat(3, {st_mode=S_IFREG|0644, st_size=1433, ...}) = 0
44. mmap(NULL, 1433, PROT_READ, MAP_PRIVATE, 3, 0)
    = 0x7f9b5f81d000
45. close(3)                                 = 0
46. openat(AT_FDCWD, "/usr/share/locale/en/LC_MESSAGES/libc.
    mo", O_RDONLY) = -1 ENOENT (No such file or directory)
47. write(2, ": No such file or directory", 27: No such file or d
    irectory) = 27
48. write(2, "\n", 1
49. )                      = 1
50. close(1)                                 = 0
51. close(2)                                 = 0
52. exit_group(1)                            = ?
53. +++ exited with 1 +++
```

代码 1–14

从这里可以看到，如果创建一个文件，前面的目录路径是不存在时，那么是会返回错误码信息的，同时，如果以后看到一些系统命令使用，发现提示 Operation not support 这样的信息时，那么就可以根据 strace 命令，来看看具体是哪一步的命令调用没有支持，又或者命令卡住了，也可以通过 strace 命令来辅助排查问题。

1.6.2 man 命令

man 命令相信大家都了解，man 命令是有助于查询 Linux 命令手册的，而一般来说，可以使用 man write 来查看 write 这个系统调用接口的情况，然而还可以使用 man 2 write 来查看该命令，感兴趣的朋友可以对比看看，二者得到的结果会有什么不同。

1.6.3 错误码

在 Linux 命令中，如 write 命令会见到一些常见的错误码，如 EAGAIN、EIO，那么对于这些命令的返回，自研文件系统的时候，需要把内部的错误信息转换为对应的错误码返回给 fuse，然后再进行对应的处理。

其中这里特别需要留意有一些错误码是可重试的，但是有一些是不需要重试的，例如 EAGAIN 错误可以进行重试，但是对于 EIO 和 EDQUOT 错误，一般就是磁盘出现异常了或者磁盘限制容量达到上限了，并且状态不可逆的时候，这时候进行重试，也是类似的错误，应及时地进行其他异常错误处理。

如果是自研的单机存储引擎，通常会自定义一些内部的错误码，这些错误码是为了细分和识别不同的场景下的信息，如磁盘异常，往往会分为磁盘数据错误、磁盘掉线等情况，而如

果单纯使用 EIO 这样的错误来展示，是很难具体感知错误问题的，同时打印到日志中也无法辅助快速排查问题，因此往往会细分不同的场景类型错误码，但是最终都会对应到 Linux 的系统调用接口错误码返回客户端，这也是错误码的对应关系转换。

1.7 文件对齐和非对齐写

文件的操作往往大家最熟悉的是增删改查，对于写入操作，其实也分为很多不同的类型，其中包括对齐和非对齐写，这二者带来的差异会使性能和技术实现上有所不同，因此本小节就来了解和学习一下文件对齐写和非对齐写的内容。

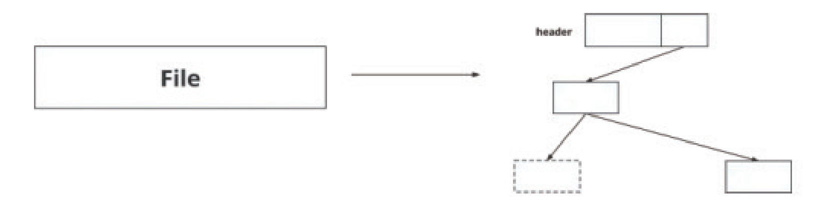


图 1–11

这是一个很基础的文件展示形式，我们通常可以把文件理解为一个很长的形式，至于里面到底是数组还是树形结构，其不同的存储系统有其独特设计。对于 Linux 文件系统来说，通常文件是树形结构的，有些是 b+ 树，有些是 avl，不管是哪种，我们通通以二叉树的形式来看待就好，具体的数据结构差异，并不影响我们对文件结构的理解。所谓的对齐读写与否，是针对存储系统的最小数据单位来说的，例如文件系统中，通常默认是 4kb，但是像 HDFS 和 ceph 这些系统，往往是以 Mb 为单位的。这里的对齐读写问题，主要指代 Linux 文件系统。也就是说，这里的读写，默认以 4kb 为单位。

如果这时候要对该数据块进行 4k 的写入，若刚好覆盖这个数据块，这种就被称为对齐了。与此相反，如果要读写的范围不是刚好覆盖这个数据块的，有可能会出现以下情况。

（1）读写跨多个数据块的，如一半在前面，一半在后面。并且都不完全覆盖多个数据块。

（2）读写在一个数据块内部的，如写入前面或者后面 2k 的数据。

以上两种情况，不管是哪种，都会面临一个问题，需要先把数据块从磁盘中加载出来，再进行写入处理。为了更好理解这个过程，我们以第二种情况，也就是写入一个数据块内部的情况为例。假设现在需要写入前 2k 的内容，那么步骤如下所示（非对齐）。

（1）从磁盘中读取该数据块，假设为 a，加载到内存中。

（2）从内存中申请一个新的 4k 的数据块，假设为 b。

（3）写入 b 中前 2k 的内容，读取 a 中后 2k 的内容，回填到 b 的后 2k 中。

（4）把 b 的内容刷新持久化到磁盘中。

这就是一次非对齐写入的过程了，那么和对齐写入的区别，其前三步会有区别，下面是对齐写入的过程。

（1）内存中申请一个新的 4k 数据块，假设为 b。

（2）直接复制数据写入 b 中。

（3）把 b 的内容持久化刷新到磁盘中。

从这两个过程就明显可以看出来，对齐写入是少了一次从磁盘加载数据块的过程，因为是完整的覆盖写，不需要关心原来的数据是怎样的。请注意，从磁盘中加载数据的过程是非常慢的，因此如果系统中有大量的非对齐写入，效率会很低，这就是为什么很多存储系统极力不推荐非对齐读写的原因之一。

对于非对齐写入，也是写入数据块的前 2k 内容，我们对原来的数据块 a 做一个假设，后面的 2k 内容，真的都有内容吗？

如果没有，或者部分没有，会出现什么问题和后果？我们来分别列举一下试试看。

1.7.1 假设一：a 的后 2k 中，没有内容

对于这种情况，我们是否可以优化一下上面的过程，如下所示。

（1）从磁盘中读取该数据块，假设为 a，加载到内存中。

（2）从内存中申请一个新的 4k 的数据块，假设为 b。

（3）写入 b 中前 2k 的内容。

（4）把 b 的内容刷新持久化到磁盘中。

区别就是在第三步中，我们并不需要再次读取 a 的内容中的后 2k 进行回填了。当然可能会有人有疑惑，这里第一步是否还有必要呢？当然有必要，如果不读取数据块 a，如何知道后面是没有内容呢？所以第一步也是没有办法省略的。那么接下来我们再来看其他假设的情况，再讨论这样会有什么问题。

1.7.2 假设二：a 的后 2k 中，只有最后的 1k 有数据

这种情况，相比起前面的内容，也是第三步的差异，就是回填 1k 的内容就好了。相信大家也不难理解。对于这两种情况，大家不知道是否发现问题所在？那就是第三步了，不管是假设一还是假设二的情况，数据块 b 中，没有回填的内容，到底是什么？答案是不知道，因为申请的一个缓存数据块空间，我们永远无法知道，之前到底里面保存了什么数据，因此对于没有填充的数据区间部分，可能会存在数据，可能会不存在数据，所以我们永远无法知道内容，这样可能会导致数据块的校验出现异常。例如假设二中的情况，a 的后 2k，只是填充了最后的 1k 数据，那么 b 前面的 1k 有数据，会导致校验不对。因此为了解决这个问题，不管哪种情况，对于申请到的新的数据块 b，都

要重新重置数据（填零），而且必须对整个数据块进行的。

1.8 文件 truncate

对于前面小节中提到的文件被提示 file truncated，表示该文件展示的内容已经被 truncated 删掉了。那么这里就产生一个小疑惑，能否再次对 truncated 的文件进行追加内容呢？例如使用 date >> xxx.txt 文件这样的操作。答案是可以的，这个小实验很简单，大家可以自行尝试一下。同时该操作也说明一个情况：truncate 并不会删除文件，否则不应该可以再次追加内容。

同时为了更加准确地知道 truncate 的含义，再次借助 Linux man 命令来获取该命令的使用，还包括了具体的参数等情况，现在先让我们来看看 man 文档中是如何描述 truncate 的，如表 1–1 所示。。

```
TRUNCATE(1)                                         User Commands
TRUNCATE(1)
NAME
    truncate - shrink or extend the size of a file to the specified size

SYNOPSIS
    truncate OPTION... FILE...

DESCRIPTION
    Shrink or extend the size of each FILE to the specified size
    A FILE argument that does not exist is created.
    If a FILE is larger than the specified size, the extra data is
lost.  If a FILE is shorter, it is extended and the extended part (hole)
reads as zero bytes.
    Mandatory  arguments  to  long  options  are  mandatory  for
short options too
```

表 1–1

这段话里面，包括了 shrink 和 extend 两个关键字，这两个单词很好理解，分别对应缩小和扩大。对于缩小，那么可能

会丢失数据（the extra data is lost）。当然不管是哪种情况，这里都没有提到文件会删除，所以还是验证了前面我们提到的情况，那就是 truncate 不会删除文件，这点很重要，千万不要把 truncate 和 rm –fr 删除命令搞混了，这是有本质区别的。

那么我们接着需要思考一个问题，什么情况下需要 truncate 呢？如果接触过数据库，如 mysql 等，可能大家使用或者听过 truncate table 的操作，这个操作往往是清空数据，但是并不删除表，这和文件的操作有点类似，但是数据库中 truncate 和文件系统里面的使用还是有区别的。

文件系统中使用 truncate 变大是为了提前把文件结构构建好。为了理解这句话，我们可以来思考一个这样的场景，如果文件的数据块是默认的 4k，那么一个 4G 的文件，文件层级会是多少呢？这个取决于文件结构的具体情况。假设每个叶子节点的父节点，可以指向 1024 个叶子节点（这已经是非常大的数量了，但实际上文件系统往往并不会有那么多，因为指向每个叶子节点的指针都需要一定的空间，上千个指针需要的空间也非常大了）。即使这样，还是需要数以千计的节点来指向叶子节点。

同时还要需要记得这样一个情况，文件在不同大小的时候，文件层级是不一样的。也就是说，如果文件刚开始写入的时候，因为数据很小，可能 level 是 1，后面会慢慢变成 2，最后可能到达 5、6 等，如图 1–12 所示。

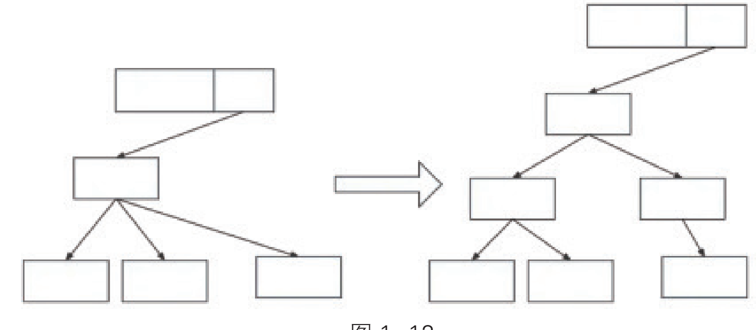


图 1–12

图 1–12 就是一个简单文件结构的变化图，每次当文件写入一定程度要进行变换 level 时，必然需要阻塞住其他的写入，因为要先变化文件结构，会改变父节点的指向（大家可以想想具体的变化情况）。那么这样会导致写入的时候有停顿，有一点类似 jvm 的垃圾回收时的“stop the world”。

因此为了解决这个问题，当文件要创建之后，可以先使用 truncate 一次性把文件指定到最后想要写入的大小，提前把文件结构创建好，然后在写入时可以并发写入，这样可以提高速度。

1.8.1truncate 变大

有了这样的需求之后，我们接下来就需要思考第二个问题了，如果使用 truncate 变大之后文件的大小显示到底应该是多少呢？因为这时候只是先提前把文件结构构件好，没有真正写入数据，但是又使用了 truncate 变大。为了解答这个问题，我们通过实际来证明一下。

```
$ stat 2.txt
  File: 2.txt
  Size: 0         Blocks: 0          IO Block: 4096   regular empty file
Device: 10303h/66307d   Inode: 2404805     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  hilton)   Gid: ( 1000/  hilton)
Access: 2023-02-26 13:45:11.491626371 +0800
Modify: 2023-02-26 13:45:11.491626371 +0800
Change: 2023-02-26 13:45:11.491626371 +0800
 Birth: -

$ truncate -s 1024 2.txt

$ stat 2.txt
  File: 2.txt
  Size: 1024      Blocks: 0          IO Block: 4096   regular file
Device: 10303h/66307d   Inode: 2404805     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  hilton)   Gid: ( 1000/  hilton)
Access: 2023-02-26 13:45:11.491626371 +0800
Modify: 2023-02-26 13:45:20.331679045 +0800
Change: 2023-02-26 13:45:20.331679045 +0800
 Birth: -
```

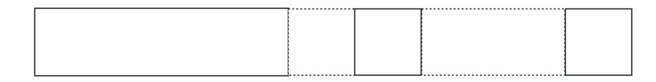
代码 1–15

truncate 变大之后的大小，就是指定的文件大小，目前并没有做任何的数据写入，因此有了第二条说明出现：truncate 变大，文件的大小会以指定的 size 为准，哪怕没有实际数据写入。

如果创建了文件之后立刻进行 truncate，这时候是一个空文件，虽然文件的 size 显示是很大的，但是会造成了一些“误解”。因为后面要写入数据的时候，可能已经隔了很久，同时文件会被关闭或者被系统重启，打开文件写入的时候，并不知道文件是否是真实数据，是否不会让文件系统去磁盘读取数据，才发现并没有真正的数据出现性能损耗。

这是一个好问题，解决这个问题的方法也是很简单，那就是可以在文件的元数据里面加上一个文件的“真实数据大小”，例如以一个“real size”来进行标记（可以以 real_size 来表示），但是这个数值并不一定会在 stat 中展示出来，是一个内部的使用字段，文件有写入之后，文件写入偏移不断往后，同时该数值也不断累加，但是不会超过文件的大小。

引入一个内容来解决问题，那么往往也会引起另外一个问题。使用了 real_size 来解决文件内部显示真实大小之后。下面有一种场景如图 1–13 所示。



File

图 1–13

上图是一个相对特殊的文件，其中虚线部分表示文件没有写入，实线部分代表文件有真实数据。这个文件的出现，可以是先创建一个文件，truncate 变大，接着写入的时候，指定偏移，然后再指定下一次的偏移量，那么写入就不会是完全连续的，会是一种写一部分，然后跳过，再写另一部分的情况（这种情况相对少见）。

这里就会产生另外一个问题了，前面提到的 real_size，到底是指向多少呢？是文件偏移量最大的那一次写入的文件大小，也就是不管中间是如何跳跃写入的，real_size 都应该记录偏移量最大的那一次写入的大小。根本原因就是偏移量最大的那一次写入决定了文件最终的结构形式。

1.8.2 truncate 变小

前面提到了文件truncate变大的情况，接着来了解一下 truncate 变小，相较于变大，变小遇到的问题会更加复杂一些。如图 1–14 所示。

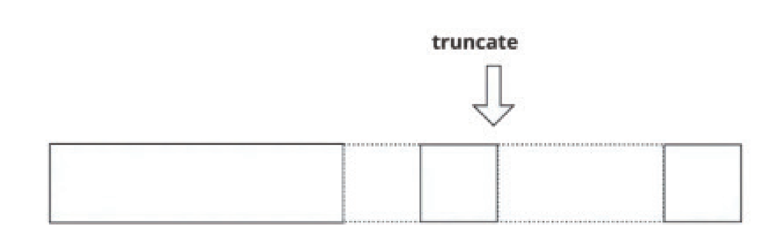


图 1–14

以前面提到的文件为例，这里如果使用 truncate 变小，那么指定一个到一个大小后，以该大小为分界线，超过该范围的文件数据会丢弃，同时会修改变小后的文件大小。那么通过这个例子，这里就又可以有一条说明：truncate 变小，会丢失指定大小后到文件原来大小范围内的数据，这是一个范围的数据丢弃。

为什么这里要加上一个范围的数据丢弃呢？这是因为 truncate 的变小，并没有办法把文件的数据一点点地丢弃，特别是适合于要进行批量丢弃删除数据的场景。一个场景就是要删除日志文件数据，有时候测试数据很多，需要在进行测试之前，清空原来的旧数据，但是并不想删掉日志文件，往往会使用 truncate –s 0 的命令来直接清空，当然使用 echo > 这样的重定向，也可以达到相同的目的。

那么 truncate 变小的时候，丢弃数据后就会产生一个问题，如果要指定大小，恰好不是对齐写的怎么办？也就是对于边界的数据块该如何处理呢？为了更好地理解这个问题，可由图 1–15 来解释。

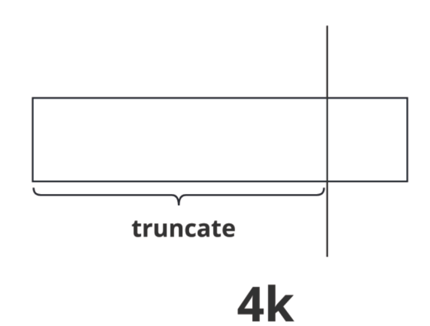


图 1–15

为了简化场景，假设现在要 truncate 变小，在最后一个数据块内部，可能会存在上述的场景，恰好 truncate 指定的大小，是该数据块内部，那么对于该数据块该如何处理呢？这里假设 truncate 到 3k 大小，其步骤如下所示。

（1）读取原来的数据块 a。

（2）内存申请一个新的同样大小的数据块 b，初始化 b（置零）。

（3）读取 a 的前 3k 内容，回填到 b 中。

（4）刷新持久化 b 的内容到磁盘中。

通过上面的步骤可以看到这其实就是一次非对齐写。所以对于 truncate 变小，往往也会伴随着非对齐写，同时很多文件系统也会把 truncate 纳入 write 的范畴，这就是其中的一个原因。

那么这里就又会产生一个问题，如果 truncate 变小需要进行一次非对齐写，如上述的场景，那么这里记录的文件大小，应该是 4k 还是实际的 3k 呢？下面来看看情况。

```
$ stat 2.txt
  File: 2.txt
  Size: 1024            Blocks: 0          IO Block: 4096
    regular file
Device: 10303h/66307d   Inode: 2404805     Links: 1
```

```
Access: (0644/-rw-r--r--)  Uid: ( 1000/   hilton)   Gid: ( 1000/
    hilton)
Access: 2023-02-26 13:45:11.491626371 +0800
Modify: 2023-02-26 13:45:20.331679045 +0800
Change: 2023-02-26 13:45:20.331679045 +0800
 Birth: -


$ truncate -s 100 2.txt

$ stat 2.txt
  File: 2.txt
  Size: 100             Blocks: 0          IO Block: 4096
    regular file
Device: 10303h/66307d   Inode: 2404805     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   hilton)   Gid: ( 1000/
    hilton)
Access: 2023-02-26 13:45:11.491626371 +0800
Modify: 2023-02-26 14:44:59.497141769 +0800
Change: 2023-02-26 14:44:59.497141769 +0800
 Birth: -
```

代码 1-16

展示的文件 size，一定都是由命令指定的，但是前面我们提到，往往可以在内部使用一个 real_size 来指定其真实大小，这里其实也应该是要使用 4k 的，而不是 3k，因为一个数据块内部只要有数据，那么就以整个数据块大小为最小单位进行计算，并不应该去考虑最小单位内部到底具体有多少数据，这往往也是一个计算大小的准确简便方法。

前面提到了一些情况的说明，下面都来归纳一下。

- truncate 不会删除文件
- truncate 不管变大变小，最终显示文件大小以命令指定 size 为准
- truncate 变小，需要丢弃指定大小后的数据

对于 truncate，可以看成是一个很特殊的写入。同时在对象存储中，往往不一定会支持该语义，因为其相对复杂的同时需求也不频繁，而且会出现很多问题，因此往往只有在文件系统中才会接触比较多。当然深入理解了 truncate 之后，会对文件结构和语义有了更加深刻的理解，不管是文件存储还是对象存储，在如何使用 truncate 的问题上会有更多的思考和把握了，只要使用得当，truncate 也是一个很好用的命令。

1.9 文件数据写入 write

对于文件的常见操作——增删改查，其中 write 是大家接触比较多的操作，而在文件系统的实现中，对于写入也有一些不同类型，同时也有一些小细节值得关注留意，下面来了解一下。

1.9.1 问题一：写入是到内存就返回还是会直接刷新到磁盘的

通常来说，数据是会写入内存中的，并不会马上刷新，因为数据落盘的性能实在太差了，不管是 ssd 还是 hdd，性能上来说还是比不上内存的。往往在文件系统中，写入的数据都是先在缓存中保存，累积到一定程度或者一定时间后，再进行刷新落盘的（像 zfs 和 xfs，通常是 30s 左右）。

如何保证文件一定是落盘的，那么这就是 fsync 命令的功能，通常会把对应的文件在内存中的数据进行落盘。同时这也意味着，对于文件数据的 write 请求，写入请求成功，并不代表数据的安全性得到保证了。同时这里就延伸出另外一个小问题了，数据写入缓存里面，那么是不是只能让客户端来发起刷新呢？一旦遇到缓存被修改的数据累积过多，那么该如何处理呢？

这个问题其实也很好解决，那就是后台定时刷新，在常见的文件系统中，这里就有定时刷新的队列，一旦达到了阈值就

会刷新进行数据持久化落盘，通常这个时间默认是 30s，不宜过晚，也不能过早。如果过晚，可能每个大文件写入了一部分数据之后，就会把内存空间用完导致溢出了；相反如果过早，那么可能数据累积太少，每次持久化的数据只有 KB 级别的话，那么落盘数据的写入性能会很差。如果以生活场景来举例，过早地刷新数据，类似于公交车在始发站中，只要有一两个乘客上车，那么就马上出发，这样可能后面有其他乘客过来乘车时，公交车已经在路上了，需要等待很长一段时间。

1.9.2 问题二：文件的写入，追加写和覆盖写有区别吗

首先明确一下内容，所谓的追加写，就是从文件的边界开始写入，例如现在文件大小是 4k，那么追加就是写在文件末尾开始写入，不会覆盖之前已经写入的内容；相反，覆盖写则是重复写入之前的空间中，可能是对齐写，也可能是非对齐写。那么这两种写入有区别吗？答案是追加写可能会改变文件结构，但覆盖写不会。

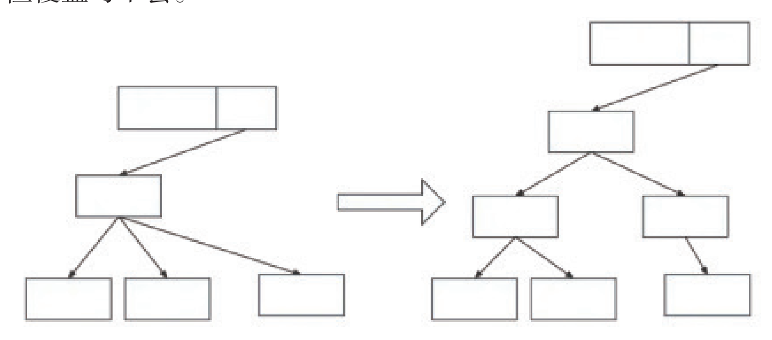


图 1–16

就以图 1–16 为例，如果是追加写，那么可能写入的偏移量的边界刚好是文件结构的变化，那么这时候就需要修改文件结构了，在 zfs 中会用文件 level 表示，而在 xfs 中则是 short 和 b+tree 结构等表示，本质上都是类似的。那么这两者的差异在

哪里呢？一旦修改文件结构，那么必然文件的元数据信息是需要更改的，同时需要申请新的数据空间来存放中间节点的信息，而不只是文件数据。

当然最重要的区别之一在于追加写会导致文件在异常修复的时候有区别。如果写入请求失败（例如文件写入数据和日志成功了，但是还没来得及修改其他信息机器就挂了），若这时候进行日志解析，则需要考虑文件结构是否发生了变化。简单来说，就是写入日志的解析需要考虑文件是否需要 truncate，因为可能之前文件是 level 1 的，但是这次写入变成了 level 2，因此在解析写入之前，需要先重新构建文件结构，这一点就和覆盖写有很大的不同，同时也是为什么 write 请求的日志解析需要考虑 truncate 的原因。

1.9.3 问题三：文件数据写入，能否做到一边写入一边异常数据修复

首先说说为什么会提出这个问题，因为在很多分布式系统里面，其实是存在文件区间锁的，其可以提高文件并发写入性能，等等。但是有一点需要注意，到底文件的区间锁是否真的有效果呢？因为从上一个问题可以知道，如果文件的写入是追加写，那么其可能会修改文件结构，但是文件结构一旦被修改，那么必然是影响到整个文件的，所以对于追加写，往往是无法让区间锁生效的。

我们知道通常在三副本的系统里面，不管是单机还是分布式的，如果有一个磁盘异常了，但是数据还是在写入，接着异常磁盘重新上线恢复的话，那么写入也在继续，这时候是该先修复异常数据还是继续写入呢？

这里分为几种情况，如果文件不存在，那么必然是要先创建文件的；但是如果文件是存在的，只是大小不一致，那么往

往牵扯到该问题。对于这种情况来说，通常是无法做到一边修改一边写入的。因为通常的文件写入，例如使用 fio 或者 cp 一个很大的文件，那么这时候的写入也是从前到后不断写入的，因此大部分都是可以看成是追加写的形式了，所以这里如果想实现一边写入一边修复就很难了。

但是并不是完全不会一边写入一边修复的，有一种情况是可以实现的：对数据进行分片。例如一个 GB 级别的文件，以 100MB 拆分数据分片，那么这时候可能是前面的分片有异常的，但是并不影响后面的数据分片，这种情况下，每个文件分片其实可以看成是一个独立的小文件，那么就可以实现一边修复一边写入了。

如果想要在文件追加写的时候实现并发，这里有多种不同的做法，一种方案是客户端先独占该文件，在写入的时候通过客户端来自行控制解决冲突问题，也就是每次写入，不会有重叠区域的追加写出现，以此来规避和解决冲突问题。

1.10 文件打开和关闭

一个文件从创建到删除过程中，可能会有反反复复地打开与关闭的过程，本小节就一起来了解一下文件打开和关闭中需要注意的内容。

1.10.1 文件打开

首先需要思考一个问题，文件打开，到底打开之后得到什么内容？如果使用 man open 命令进行查看，open 命令返回的是一个文件句柄，这里可以使用 lsof 命令来查看具体的情况，下面来简单展示一下。

```
1.  # stat /tmp/1.txt
2.    File: /tmp/1.txt
3.    Size: 0             Blocks: 0          IO Block: 4096   r
    egular empty file
4.  Device: fc01h/64513d    Inode: 3219        Links: 1
5.  Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
6.  Access: 2023-03-22 20:30:22.887597590 +0800
7.  Modify: 2023-03-22 20:30:22.887597590 +0800
8.  Change: 2023-03-22 20:30:22.887597590 +0800
9.   Birth: -
10.
11. # tail -f /tmp/1.txt
```

代码 1-17

首先创建一个文件 1.txt，接着使用 tail –f 命令来一直打开该文件，同时在另外一个终端使用 ps 命令得到其 pid 并且使用 lsof 命令来查看，如下所示。

```
1.  # ps uax |grep 1.txt
2   root    46394  0.0  0.0   7264   580 pts/2    S+   20:30   0:00 tail -f /tmp/1.txt
3.
4.  # lsof |grep 46394
5.  tail  46394  root    3r      REG          252,1          0       3219 /tmp/1.txt
```

代码 1-18

这里就是一个比较简单的打开文件的情况，通过上面可以看到 lsof 中展示该文件是 REG，也就是文件类型，同时 3r 表示只读。

实际除了以上内容，一个文件被打开的时候，还需要考虑什么呢？其中一点就是该文件是否有异常数据需要进行修复。因为在 zfs 中有存储池的设计，数据可能会保存在多个不同的磁盘中进行备份，因此一旦遇到文件异常了，就要触发修复流程。而在分布式系统中，文件是否触发修复，一般通过文件元数据中心保存的信息来判断，而 glusterfs 则是通过文件的扩展属性来进行判断，该操作被称为 AFR。

文件的打开还需要留意一个细节，如果文件数据很多，例如大小达到了上百吉字节，这时候的文件结构通常是树形结构的，同时部分文件系统采用 extend 结构，会有非常多的数据元信息需要加载，因此在首次打开文件的时候，是一次性加载还是部分加载则是一个需要思考的问题。如果一次性加载大量的数据信息，那么必然会导致文件打开所需时间很长，因此在 ext4 等文件系统中，会对文件 extend 信息加载有一些限制和优化，具体可以见源码函数 ext4_find_extend 的细节。

1.10.2 文件关闭

相对于文件打开操作，文件的关闭则比较复杂一些，因为在正常情况下，文件打开都是在系统正常情况下进行的，如果系统异常，那么文件打开失败也不会影响文件的数据。但是文件的关闭，往往情况比较复杂，包括文件正常关闭、异常关闭等情况。其中文件正常关闭的时候，还要考虑文件中是否有已经写入缓存的脏数据，如果有脏数据，必须先等待脏数据持久化到磁盘中才能真正关闭，否则会存在丢失数据的情况。因此一旦文件发起关闭操作，可能并不会立刻关闭，有可能会在系统有异步操作中进行。

同时文件正常关闭往往也会分为客户端主动触发关闭和被动关闭两种情况，其中主动关闭通常是客户端在写入数据并且刷新数据之后，保证数据已经持久化到磁盘后调用关闭操作。而被动关闭，则是客户端由于网络等原因，导致客户端与服务端之间连接中断，但是文件之前一直处于被打开的状态，为了不让文件长时间没有操作占用缓存空间，因此通常情况下，服务端节点会根据时间判断，在一定时间周期后，若文件没有接受到任何用户请求，那么会触发被动关闭操作。该操作其实和缓存数据周期性持久化类似，不管文件的引用基数如何，因长

时间没有操作，则认为该文件可以关闭，但是关闭之前也需要先把缓存中文件的修改数据持久化到磁盘中，保证数据安全。

1.11 文件属性信息

每个文件除了数据，还会包含非常多的属性信息，其中属性信息还分为文件元数据信息和扩展属性，前者是每个文件包括目录（目录可以被认为是一种特别的文件类型）等创建之后都会包含的，而扩展属性则是在一些场景下，用户通过命令进行创建的。下面来了解一下关于文件属性信息的相关内容。

1.11.1 问题一：文件的三个时间什么时候会改变

每个文件在创建之后，使用 stat 命令都会显示三个时间，分别是 access、change 和 modify，那么这三个时间都会在什么情况下进行改变呢？下面做一个简单的小实验来了解一下。

```
1.  # touch /tmp/a.txt
2.
3.  # stat /tmp/a.txt
4.    File: /tmp/a.txt
5.    Size: 0         Blocks: 0       IO Block: 4096   regular empty file
6.  Device: fc01h/64513d    Inode: 3776         Links: 1
7.  Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
8.  Access: 2023-03-25 13:46:49.710174458 +0800
9.  Modify: 2023-03-25 13:46:49.710174458 +0800
10. Change: 2023-03-25 13:46:49.710174458 +0800
11.  Birth: -
12.
13. # date >> /tmp/a.txt
```

```
14.
15. # stat /tmp/a.txt
16.   File: /tmp/a.txt
17.   Size: 29        Blocks: 8       IO Block: 4096   regular file
18. Device: fc01h/64513d    Inode: 3776        Links: 1
19. Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
20. Access: 2023-03-25 13:46:49.710174458 +0800
21. Modify: 2023-03-25 13:47:01.050513512 +0800
22. Change: 2023-03-25 13:47:01.050513512 +0800
23.  Birth: -
```

代码 1-19

上面的操作是先创建文件，再追加了一条信息，接着查看文件属性信息，可以发现三个时间中，modify 和 change 两个时间发生改变了，只有 access 时间并没有改变。接着我们再继续做一个实验。

```
1.  # cat /tmp/a.txt
2.  Sat Mar 25 13:47:01 CST 2023
3.
4.  # stat /tmp/a.txt
5.    File: /tmp/a.txt
6.    Size: 29         Blocks: 8       IO Block: 4096   regular file
7.  Device: fc01h/64513d    Inode: 3776        Links: 1
8.  Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
9.  Access: 2023-03-25 13:47:18.759042355 +0800
10. Modify: 2023-03-25 13:47:01.050513512 +0800
11. Change: 2023-03-25 13:47:01.050513512 +0800
12.  Birth: -
```

代码 1-20

从这里可以看到，打开查看文件之后就会改变 access 的时间了，对于大部分存储系统来说，读多写少的场景下，如果频繁修改 access 时间，那么会造成不可忽略的性能损耗，因此为了优化该问题，在 nfs 和 zfs 中，都可以通过参数来关闭 access 时间的频繁更新。例如在 nfs 中，可以通过 -o noatime 选项来进行优化。除了这个参数，mount 的时候还有一个 nodiratime 也是

可选的。

1.11.2 问题二：文件创建和修改时，上层目录的属性信息都会改变吗

有了对前面问题的了解之后，接下来我们再来延伸一下，如果该文件在一个目录层级相对较多的情况下，那么当文件被修改之后，会更新其父目录的信息吗？会更新其父目录之上的所有上级目录信息吗？会不会一直更新到根目录呢？对于这个问题的解答，我们还是可以以一些简单的小实验来验证一下。这次的实验可以先创建多个目录，接着先使用 stat 查看每个目录的信息，再在最底层目录下创建文件和写入数据，再次使用 stat 查看目录的信息对其进行对比，这样就可以得到答案了。因此关于该问题的答案，留待各位自行实验后得到。

1.11.3 问题三：文件属性信息异常是如何修复的

关于该问题，我们先来查看一些场景，再结合该问题进行思考。目前都是在三副本的场景下进行的。假设三副本的节点或者磁盘分别用 a，b，c 来进行表示。

➢ 场景一：数据重复完整写入异常

该场景构造步骤如下所示。

（1）带有数据的文件写入三节点中。

（2）关闭节点 c 后，删除文件。

（3）再次重复写入文件。

（4）重启节点 c。

➢ 场景二：空文件扩展属性不一致

对于该场景的构建与上面场景类似，但是需要创建一个空文件，同时在节点关闭后，通过 setfattr 来增添一条文件的扩展属性。该场景还可以通过先创建空文件，在节点关闭后，对

其他两个节点的空文件进行写入数据，再删除清空数据来构建场景。

通过这两个小的场景，我们可以发现，文件的数据是一致的，如果按照直觉理解，可能会认为文件并没有异常，但是存在文件的属性信息不一致，那么这时候修复该如何进行呢？其实文件的属性信息也可以认为是一种数据，按照文件的数据形式，通常以多数一致原则来修复即可。但是通常存在一个容易忽略的点，如果在自研的存储系统中，要考虑这些空文件的属性异常场景，可以通过添加单元测试的方式来保证系统的稳健性。

2、深入理解zfs

2.1 zfs 存储池设计

zfs 虽然是很有名的一个文件系统，但是除了 zfs，linux 系统中最常见的默认文件系统应该是 ext4，还有 btrfs 和 xfs 也是主流的文件系统，那么已经有了这些文件系统之后，为什么还要去关注和设计一款新的文件系统，为了理解这个问题，还需要了解一下 zfs 当初诞生的背景。

zfs 最早在 2001 年就开始设计了，是 sun 公司存储团队的负责人 Jeff 带领团队设计并研发的，后被 oracle 收购，并且 zfs 有一个开源的版本叫 openzfs，后面如无特殊说明，zfs 都是默认指代 openzfs。而 zfs 受欢迎的原因之一，是 zfs 被称为最后一代文件系统，其设计之初就考虑了在软件层面实现磁盘冗余（以前通常使用 RAID 硬件卡来实现，但是有额外的物理硬件成本，并且硬件还会受到厂商和硬件兼容限制等），因此直到现在，很多中小公司在考虑单机多磁盘层面实现存储数据冗余层面的话，zfs 都是一个非常好的选择。而且 zfs 在磁盘管理方式的实现方面很友好，操作性也非常简便。

另外在 zfs 的设计中，还有非常多有趣且独特的设计，并且这些设计内容也影响至今，如空间碎片化优化、重复数据删除等技术，早期的文件系统实现中，很少有考虑这些技术的，而 zfs 在 2009 年前后就考虑到了，关于这些技术的内容，后面也会有内容专门提到。

在 zfs 的设计中，早期就把寻址空间的位数设置为 128 位，这样就意味着 zfs 可以支持的存储容量上限是非常高的（目前暂时没有能达到容量上限瓶颈），支持 ZB 级别的存储容量空间。

现如今，zfs 很可惜的是没有办法并入 linux 的分支代码中，一大原因是早期的开源协议问题。不过如果想要使用 zfs，目前 ubuntu 操作系统在安装的时候也都可选支持了。

另外，zfs 的操作文档一般是英文的，oracle 官方有一些 oracle zfs 的文档，各位在学习和了解 zfs 的过程中可以参考官方文档，[5] 或者自行搜索 Oracle ZFS Administration Guide 相关内容进行学习。

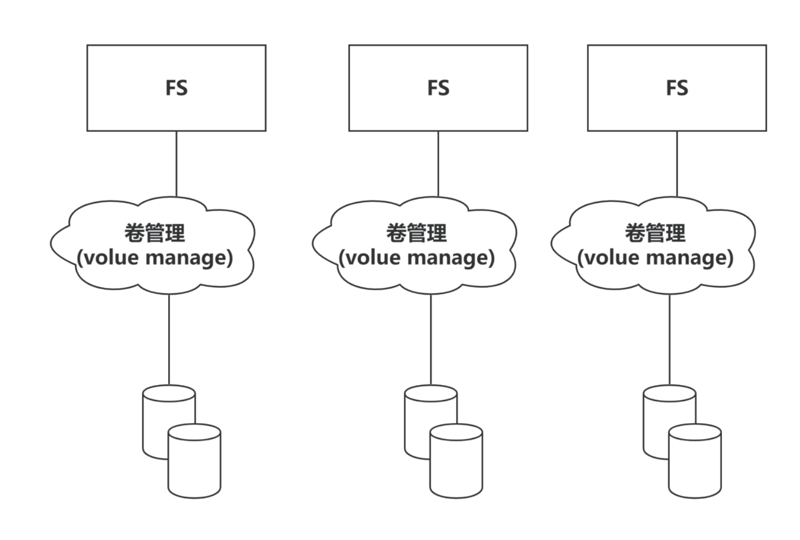


图 2–1

zfs 一个强大的功能就是 storage pool 概念，也就是存储池，这个功能可以实现软 raid 功能，而在传统的磁盘管理下，磁盘分区管理如图 2–1 所示，这里每个磁盘上面单独有一个卷管理，格式化后交给文件系统使用，zfs 的设计则有些不同，见图 2–2 所示。

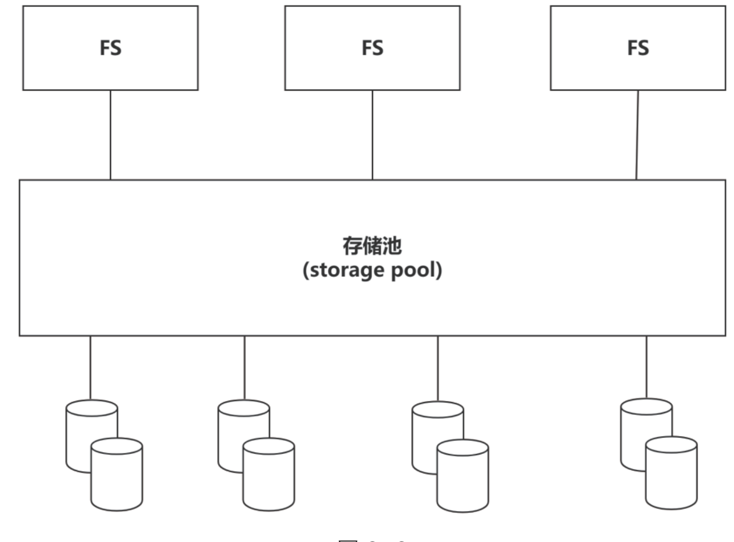


图 2–2

相比起传统的磁盘管理方式，文件空间的使用管理也是一种非常麻烦的方式，因为可能有些磁盘已经使用很多内存了，但是其中一些使用很少的内存，导致出现数据倾斜的问题，甚至会导致其中一个磁盘分区爆掉引发数据丢失的风险，等等，而 zfs 的磁盘存储池管理方式，则可以把底层所有的磁盘看作是一个很大的池子，用户不需要关心底层的分配逻辑，而数据在写入的时候就可以自动均衡地分配在不同的磁盘上（当然这也取决于用户设计的磁盘冗余程度，也就是 raidz 级别）。

这里需要说明一下，可能会有人把 zfs 的存储池设计与 Linux 的 lvm 设计弄混，这二者其实是有很大区别的，因为 zfs 的存储池设计是由 zfs 文件系统来管理的，而 lvm 是由 Linux device–mapper 提供的。同时 Lvm 中使用的卷组概念，看起来和 zfs 很相似，但是实际上，在文件系统写入数据的时候，并没有办法真正知道数据存储在哪个磁盘上，因此一般文件系统是建立在 linux 已经创建好的 lvm 的卷组上的。而 zfs 是可以自行控

制的，这二者有着本质区别。另外 Lvm 需要在安装的时候就指定好，否则后续更改是非常麻烦的，而 zfs 的存储池管理，在磁盘上下线的要求中，没有 lvm 那么复杂和繁琐。另外 zfs 还支持 hot spare，也就是热备盘，一般存储池的某个磁盘有问题，可以替换并转移数据。

在提到 raidz 之前，需要先说明一下 raid 的几个级别的区别的，方便后续做对比。通常使用比较多的是 raid 0，1，5 和 6。

raid 5 和 6 的区别就是，raid 5 只是对数据做校验，得到一个校验带数据，而 raid 6 则是两个，因此 raid 5 级别只允许损坏一个磁盘，raid 6 则是两个。而 raid 0 则没有冗余，只是单纯地将数据条带化，也就是对数据按照一定大小平均切分，再分别存放在不同的磁盘上，可以通过多磁盘的 IO 来增加性能，但是没法在单节点层面保障数据的安全性。（这里提到注明单节点，那么肯定在多节点上，分布式环境下是可以保证的，关于这个话题后续也会详细讲解一下，其涉及从软件到硬件层面的一个数据冗余保障级别问题）。raid 1 则是做镜像，把两块磁盘类似主从那样做镜像。

```
1. # add-apt-repository ppa:jonathonf/zfs
2. # apt install zfsutils-linux zfs-dkms
3.
4. # zfs --version
5. zfs-2.1.5-1~20.04.york0
6. zfs-kmod-0.8.3-1ubuntu12.13
```

代码 2-1

本机是使用 ubuntu20 来进行测试的，如果直接安装 zfsutils-linux 的版本则是非常低的旧版本，因此需要添加 ppa 获取最新版本来安装。对于后续提到的 zfs 的版本，如果没有特殊说明，默认都是使用 2.X 版本进行。

使用 raid5 需要三块磁盘，raid6 是四块磁盘。zfs 中的 raidz 也是类似 raid 级别的，其中 raidz 则是和 raid5 类似，raidz2 则是和 raid6 类似，而 raidz3 则是允许最多三块磁盘损坏。

```
1. ~$ sudo lsblk
2. sdb      8:16   0     5G  0 disk
3. sdc      8:32   0     6G  0 disk
4. sdd      8:48   0     8G  0 disk
5. sde      8:64   0     7G  0 disk
6. sdf      8:80   0     9G  0 disk
7. sdg      8:96   0    10G  0 disk
```

代码 2-2

这里是使用虚拟机创建了六块大小不同的磁盘，再使用 zpool 命令来创建存储池。

```
1. ~$ sudo zpool create hilton_raidz2 raidz2 sdb sdc sdd sde sdf spare sdg -f
2. ~$ sudo zpool status
3.   pool: hilton_raidz2
4.  state: ONLINE
5. config:
6.
7.         NAME             STATE     READ WRITE CKSUM
8.         hilton_raidz2    ONLINE       0     0     0
9.           raidz2-0       ONLINE       0     0     0
10.            sdb          ONLINE       0     0     0
11.            sdc          ONLINE       0     0     0
12.            sdd          ONLINE       0     0     0
13.            sde          ONLINE       0     0     0
14.            sdf          ONLINE       0     0     0
15.        spares
16.          sdg            AVAIL
```

代码 2-3

这里注意，生产环境不建议使用 /dev/sdx 这样的路径来创建 zpool，防止因为磁盘盘符漂移到最后节点重启后出现问题。盘符漂移就是如原来 sda 的盘符变成了 sdc。这里如果想要演示磁盘替换，则可以卸载其中一块正在使用的磁盘即可。生产环境可以使用 uuid 或者对磁盘打标签的方式来进行操作。

通过这样一个简单的命令，也可以看出 zfs 的存储池管理比 lvm 简单很多，因为 lvm 中复杂的 vg 和 lv 等概念，还有各种复杂的参数，对于大部分人来说都不是容易操作的，而且非常容易出错。

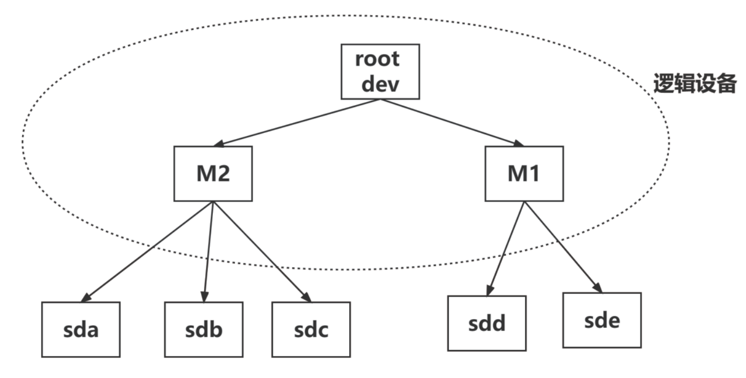


图 2–3

前面只是从特性和使用方面来描述 zfs 的存储池设计，那么在其内部设计是如何实现的呢？这里可以参考图 2–3 所示。zfs 的存储池是由虚拟设备组成的，这里有两种类型的虚拟设备，分别是物理虚拟设备和逻辑虚拟设备。一个物理虚拟设备，是一个可写入的磁盘或者块设备等，而一个逻辑虚拟设备则是管理和抽象物理虚拟设备的，也就是图 2–3 中的 M1 和 M2。

如果这里是一个 mirror pool 的话，写入数据的时候，逻辑虚拟设备就会把写入请求分发到其所有的子设备，但是读取的时候，只需要读取其中一个设备即可。那么这里如果是条带方式

进行管理的话，写入的时候也可以增加磁盘的写入带宽，因为会把数据分片写入不同的磁盘当中（当然这里不是简单的分片，像 raid5 那样的，这里的内容是另外一方面的，主要会涉及一些数学方面的内容，后续再介绍）。另外这里还有一个叫 top–level vdevs 的概念，也就是 root vdevs 下的间接 vdev，这里 top–level vedevs 的意思，可以理解为几个磁盘组成的一个池子，也就是不同的 zpool 的内容，如图 2–3 中 sdd 和 sde 组成的一个镜像组。

对于底层不同磁盘组成的存储池，这里早期 zfs 还有一个独特设计（这里可以参考 ZFS On–Disk Specification 这篇资料），叫作 vdev labels。这里的“v”是指 virtual 的意思，对应的就是图 2–3 中的 M1 及其物理磁盘，都需要记录一下该存储池中的信息。

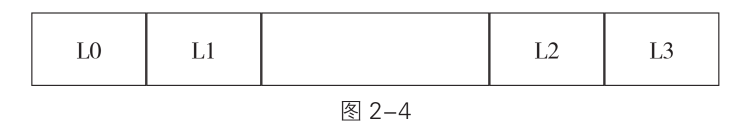


图 2–4

对于存储池的每份信息，这里会保存四份，分别为 L0 ~ L3，而且在头尾各保存两份。当需要修改的时候，先修改一头一尾的两份数据，如果中间出现问题了，那么可以通过另外两份来回滚数据，而直到四份数据都更新完成才算结束，因为这里保存的信息关系到整个存储池的核心元信息，因此通过这样的方式可以保证数据更新安全。

那么每一份标签中记录的内容有什么呢？其中包括当前该存储池的拓扑结构信息、存储池的名称，还有和事务相关的信息等。vdev label 的信息被分成了四部分：8K 的空闲空间（Blank Space），8K 的 boot header 信息，112K 的 name–value 键值对以及 128 长度的 1K 大小的 uberblock 结构数组。其中 uberblock 是访问这个存储池的入口。uberblock 是访问 pool 中数据的入口。任意时刻，只有一个 uberblock 处于激活状态，所有 uberblock 中事务组编号最高且通过 SHA–256 checksum 验证合法的 uberblock

为激活的 uberblock。

下面看一下 zfs 中 uberblock 和 object set 的关系示意图，如图 2–5 所示，在 zfs 中 object set 和 Linux 中的一切文件思想都是相似的。每个模块都可以看成不同的对象（object），而 objset 就是对象集合。

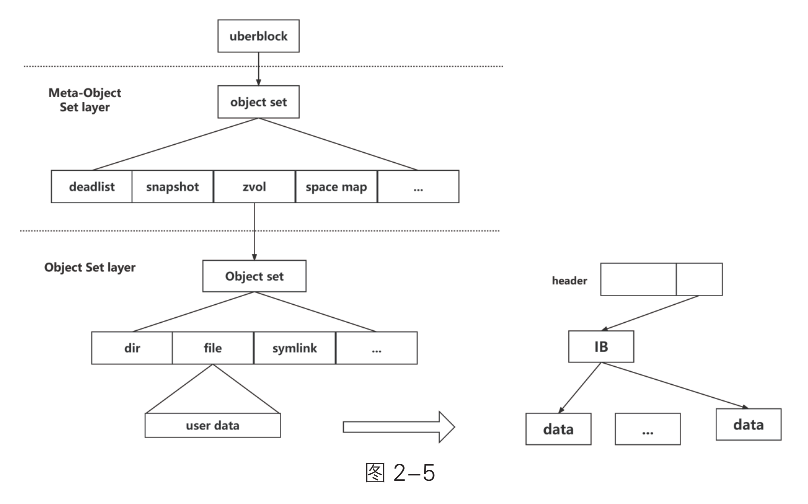


图 2–5

图 2–5 中的 Meta–Object，简称 MOS。snapshot 就是快照，而当快照被删除时，其中的数据则与 deadlist 有关，记录一些要被删除的数据信息。

zvol 全称是 zfs volume，是 zfs 向外提供与块设备相关的模块，如果想创建一个块设备给其他服务使用，可以使用 zfs create 命令进行创建，然后该块设备可以被 isici 格式化与挂载，对接 k8s 的 pvc 进行数据存储。

space map 是 zfs 的空间管理模块 metaslab 中的日志信息，关于该内容后面小节会详细讲解。

2.2 zfs 层次结构

前面提到过 Linux 的文件系统层次，但是那个对于文件系统内部结构只是一个简单的图示表示，对于文件系统内部的模块并没有深入分析过，而且日常中很多非底层研发的朋友，可能对于文件系统来说，如 ext4 和 xfs 这些，都会觉得差不多，或者会对研究文件系统内部模块感到困惑。对于这些问题，笔者觉得，有机会的话，深入以后再回头看，会有更深的体会，若将文件系统和其他领域结合起来，未来大有可为。

不管怎样，我们需要找一个常用的流行的文件系统来深入理解和学习一下，而 zfs 则是一个非常好的选择，作为号称最后一代的文件系统，我们接下来会深入研究一些模块及其原理。

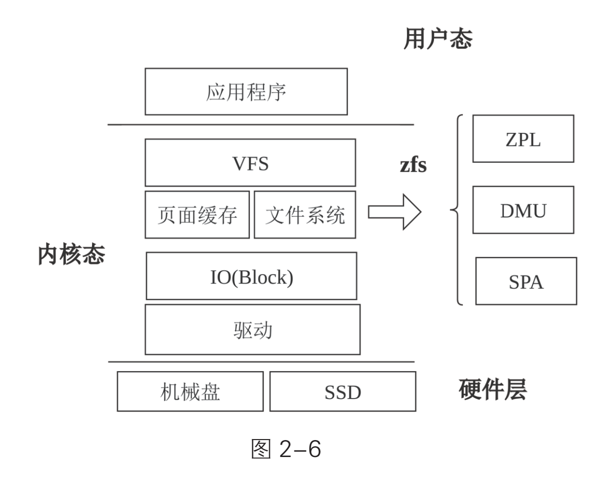


图 2–6

图 2–6 右侧部分就是 zfs 的简单架构示意图，从下到上，这里 spa 全称为 storage pool allocator，也就是存储池分配器，从名称中就可以看出，spa 就是对应最底层的磁盘空间申请和分配的。SPA 处理块分配和 I/O，并且对接上一层的 DMU。

这里 spa 既然是负责底层存储池分配的，那么可想而知，对于多个磁盘组成的存储池，写入磁盘的时候需要考虑哪些内容呢？还有一旦某个磁盘发生错误了，热备盘数据的迁移等功能，都应该由 spa 来具体负责吗？

2.2.1DMU

DMU 全称为 Data Manage Unit，叫作数据管理单元，DMU 将虚拟地址转换为从 SPA 到事务对象的 ZFS POSIX 层（ZPL）的接口，最后，ZPL 在 DMU 对象之上实现 POSIX 文件系统，并将文件的操作反馈到系统调用层。简单来说，当数据从应用中写入时，经过 VFS 然后到达 DMU 的时候，需要对数据做事务性处理，例如为了保障数据的安全，目前业务写入的数据通常采用 copy-on-write 方式进行处理，还有写入的数据要进行校验码计算等功能，这些也是在 DMU 中进行的。同时如果系统中的文件数据开启了快照等功能，那么这些事务的处理是放在了 DMU 中进行的，而 spa 则是把数据从磁盘中读取出来，并且写入磁盘落盘。

同时不管是读写请求，如果这里有压缩或者加密的需要，其处理的流程也是在 DMU 中处理。而读取数据的时候，DMU 会封装 zio 请求，先从缓存中获取，如果无法获取到，则从磁盘中加载，也就是封装成 bio 发送到内核中。关于读请求的过程，会在后面讲解。

2.2.2 ZPL

ZPL 全称为 zfs posix layer，既然是和 posix 有关的，那么显而易见这里就是对 posix 系统调用接口的实现，因为前面提到每个文件系统都可以实现 FOP，也就是 file opertation，对于一些文件的系统调用接口，都是交给注册的文件系统来实现的，因此

ZPL 就是在做这个事情。

当然对于这三个层次，并不是 zfs 的全部层次结构，只是这个结构是早期 zfs 实现的时候，最先考虑的。而像缓存 L2ARC 的内容实现，则是比较靠后的事情了，包括使用 ssd 作为日志的高速缓存等功能实现，也是后面才考虑的。

下面来看一个更加详细的 zfs 层次子模块图，如图 2–7 所示。

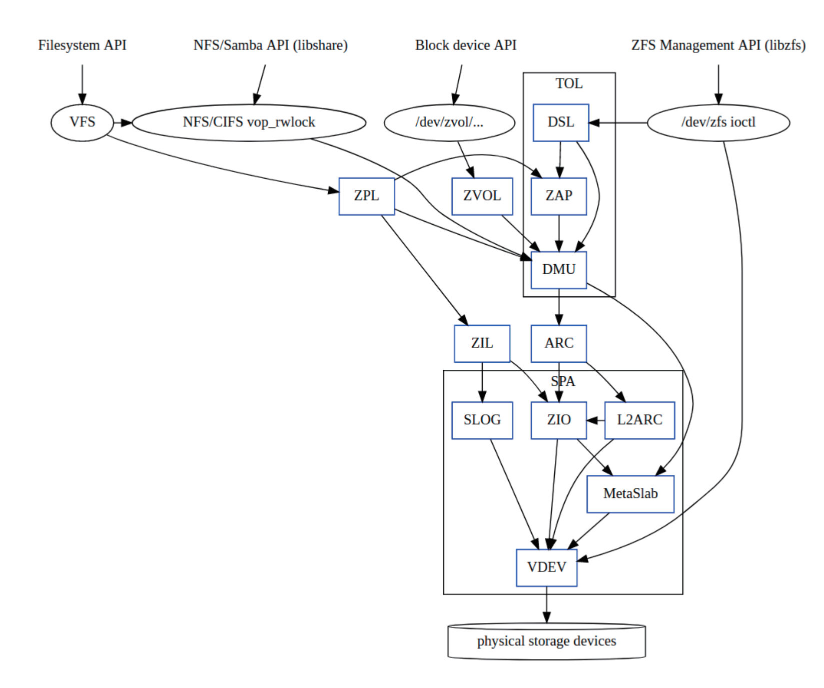


图 2–7 zfs 层次子模块图 [12]

对于 zfs 来说，对外是提供了 zpl 和 zvol 的，分别是文件和块设备的实现，而 zvol 则可以创建一个块设备，在生产环境中使用 iscsi 进行格式化，接着挂载到 k8s 中给 pvc 进行使用。

ZAP（zfs attribute processor）则是与属性相关的内容，文件中除了数据，还有一些固定的元数据信息和扩展属性，这些内容都是需要存储的，关于这部分内容在后面小节会详细讲解。

Spa 中有 slog 和 L2ARC 这些，都是与缓存相关的，这些内容也会在后面进行讲解。DSL（Dataset and Snapshot layer），根据名称可以知道，这里就是与数据快照有关的内容。

2.3 zfs 指针结构和文件结构

数据以块为单位在磁盘和内存之间传输。块指针（blkptr_t）是一个 128 字节的 zfs 结构，用于物理定位、验证和描述块磁盘上的数据，在 zfs 中被称为 blkptr_t，下面来简单看一下该结构中的内容。

```
1.  * 64 56 48 40 32 24 16 8 0
2.  * +-------+-------+-------+-------+-------+-------+-------+-------+
3.  * 0 | pad |  vdev1          | GRID  | ASIZE  |
4.  * +-------+-------+-------+-------+-------+-------+-------+-------+
5.  * 1 |G|   offset1    |
6.  * +-------+-------+-------+-------+-------+-------+-------+-------+
7.  * 2 | pad |  vdev2          | GRID  | ASIZE  |
8.  * +-------+-------+-------+-------+-------+-------+-------+-------+
9.  * 3 |G|   offset2    |
10. * +-------+-------+-------+-------+-------+-------+-------+-------+
11. * 4 | pad |  vdev3          | GRID  | ASIZE  |
12. * +-------+-------+-------+-------+-------+-------+-------+-------+
13. * 5 |G|   offset3    |
14. * +-------+-------+-------+-------+-------+-------+-------+-------+
15. * 6 |BDX|lvl| type | cksum |E| comp|   PSIZE  |    LSIZE  |
16. * +-------+-------+-------+-------+-------+-------+-------+-------+
17. * 7 |  padding     |
18. * +-------+-------+-------+-------+-------+-------+-------+-------+
19. * 8 |  padding     |
20. * +-------+-------+-------+-------+-------+-------+-------+-------+
21. * 9 |  physical birth txg  |
22. * +-------+-------+-------+-------+-------+-------+-------+-------+
23. * a |  logical birth txg  |
24. * +-------+-------+-------+-------+-------+-------+-------+-------+
25. * b |  fill count   |
26. * +-------+-------+-------+-------+-------+-------+-------+-------+
27. * c |  checksum[0]    |
28. * +-------+-------+-------+-------+-------+-------+-------+-------+
29. * d |  checksum[1]    |
30. * +-------+-------+-------+-------+-------+-------+-------+-------+
31. * e |  checksum[2]    |
32. * +-------+-------+-------+-------+-------+-------+-------+-------+
33. * f |  checksum[3]    |
34. * +-------+-------+-------+-------+-------+-------+-------+-------+
```

代码 2-4

下面来简单分享一下部分字段的含义，如表 2-1 所示。

表 2-1

字段	含义
vdev	虚拟设备 id，zfs 的存储池设计，指定当前是哪个磁盘设备
offset	vdev 的偏移量
GRID	RAID-Z 相关的信息，目前没有使用，保留日后使用
comp	和压缩有关的功能，目前 zfs 支持五种压缩的算法选择，包括 gzip，lzjb，lz4，zle 和 zstd
checksum	用来存放校验计算结果的，256 位，这里结果代表的含义由 cksum 来表示
cksum	数值不同代表不同含义，如 4 和 gang header 等，通常用来表示文件数据的校验
lsize	逻辑大小
psize	物理大小
asize	分配大小
type	DMU object 类型，具体看 dm_object_type 这个枚举字段
phys birth txg	txg 事务组 id，和 transaction group 有关。两者通常情况下是一致的。但如果使用了 dedup 或者有磁盘移除等情况，出有不同
logical birth txg	
E	Endian，大小端模式
padding	保留以后使用的空间

对于 psize，asize 和 lsize 这三个数值，如果文件数据开启了压缩，或者有 Raid-Z，那么可能会导致出现不一致的情况。目

前最新的指针结构和以前 zfs on disk format 资料里面的已经有不一样的内容了，大家如果感兴趣的话，可以对比着学习查看。同时 zfs 中还有加密的内容，对于加密的指针结构又有一些不同，主要是多了一些不同含义的字段，其中包括加密的 key 和 salt 等，本小节内容暂时不涉及这部分内容，感兴趣的可以自行查看 spa.h 文件中的内容。

另外这里为什么会出现三份 vdev，offset 呢？因为对于一些重要的数据，尤其是 zfs 中 MOS 部分的重要内容，一般会采用三份备份，而用户数据，可能只会使用一份，不考虑冗余。

有了指针结构的内容以后，下面来简单介绍一下对象结构，也就是 Object，在 Linux 中有一切皆文件的说法，而 zfs 中则是一切皆对象，二者的思想是相似的。而文件也是对象其中的一种类型，通常就是 dnode 结构。除了文件，还有一些其他类型的 object 类型，下面展示其中部分内容。

```
1.  typedef enum dmu_object_type {
2.   DMU_OT_NONE,
3.
4.   DMU_OT_OBJECT_DIRECTORY,
5.   DMU_OT_OBJECT_ARRAY,
6.   DMU_OT_PACKED_NVLIST,
7.   DMU_OT_PACKED_NVLIST_SIZE,
8.   DMU_OT_BPOBJ,
9.   DMU_OT_BPOBJ_HDR,
10.
11.  DMU_OT_SPACE_MAP_HEADER,
12.  DMU_OT_SPACE_MAP,
13.
14.  DMU_OT_INTENT_LOG,
15.
16.  DMU_OT_DNODE,
17.  DMU_OT_OBJSET,
18.
19.  DMU_OT_DSL_DIR,
20.  DMU_OT_DSL_DIR_CHILD_MAP,
```



```
21.  DMU_OT_DSL_DS_SNAP_MAP,
22.  DMU_OT_DSL_PROPS,
23.  DMU_OT_DSL_DATASET,
24.
25.  DMU_OT_ZNODE,
26.  DMU_OT_OLDACL,
27.  DMU_OT_PLAIN_FILE_CONTENTS,
28.  DMU_OT_DIRECTORY_CONTENTS,
29.  DMU_OT_MASTER_NODE,
30.  DMU_OT_UNLINKED_SET,
31.
32.  DMU_OT_ZVOL,
33.  DMU_OT_ZVOL_PROP,
34.
35.  DMU_OT_PLAIN_OTHER,
36.  DMU_OT_UINT64_OTHER,
37.  DMU_OT_ZAP_OTHER,
38.
39.  DMU_OT_ERROR_LOG,
40.  DMU_OT_SPA_HISTORY,
41.  DMU_OT_SPA_HISTORY_OFFSETS,
42.  DMU_OT_POOL_PROPS,
43.  DMU_OT_DSL_PERMS,
44.  DMU_OT_ACL,
45.  ...
46. }
```

代码 2-5

从上面展示的内容可以看到，常见的 dnode 文件结构，zap 还有前面提到的 space map 等都是一种对象类型，还有与 zvol 块设备相关的，因此从这个枚举类型也可以进一步地感受到抽象统一的魅力。

从上面的枚举可以知道，dnode 也是其中一种 object 类型，而前面章节提到的文件不同 level 层级，从 level 0 到 level 1 再到大于 1 的情况，对于一个文件的结构的变化，这里可以了解一下 dnode_phys_t 结构，如下所示。

```
1.  typedef struct dnode_phys {
2.   uint8_t dn_type;
3.   uint8_t dn_indblkshift;
4.   uint8_t dn_nlevels;
5.   uint8_t dn_nblkptr;
6.   uint8_t dn_bonustype;
7.   uint8_t dn_checksum;
8.   uint8_t dn_compress;
9.   uint8_t dn_flags;
10.  uint16_t dn_datablkszsec;
11.  uint16_t dn_bonuslen;
12.  uint8_t dn_extra_slots;
13.  uint8_t dn_pad2[3];
14.
15.
16.  uint64_t dn_maxblkid;
17.  uint64_t dn_used;
18.
19.
20.  uint64_t dn_pad3[4];
21.
22.
23.  union {
24.   blkptr_t dn_blkptr[1+DN_OLD_MAX_BONUSLEN/sizeof (blkptr_t)];
25.   struct {
26.    blkptr_t __dn_ignore1;
27.    uint8_t dn_bonus[DN_OLD_MAX_BONUSLEN];
28.   };
29.   struct {
30.    blkptr_t __dn_ignore2;
31.    uint8_t __dn_ignore3[DN_OLD_MAX_BONUSLEN -
32.        sizeof (blkptr_t)];
33.    blkptr_t dn_spill;
34.   };
35.  };
36. } dnode_phys_t;
```

代码 2-6

由结构体的内容可以知道有动态变化的主要就是 union 联合体中的内容，而根据源码中的注释内容可以清晰的知道，这里主要有三种变化，如下所示。


```
1.    0       64      128     192     256     320     384     448 (offset)
2.    +---------------+---------------+---------------+-------+
3.    | dn_blkptr[0]  | dn_blkptr[1]  | dn_blkptr[2]  | /     |
4.    +---------------+---------------+---------------+-------+
5.    | dn_blkptr[0]  | dn_bonus[0..319]                      |
6.    +---------------+-----------------------+---------------+
7.    | dn_blkptr[0]  | dn_bonus[0..191]      | dn_spill      |
8.    +---------------+-----------------------+---------------+
```

代码 2-7

其中第一种情况则对应了前面章节中提到的文件的 header 头部保留的三个指针，用来直接指向数据块的情况，也就是文件 level 0 的情况。而其他的两种场景都有涉及 bouns buffer，这个是和属性有关的，不只是文件属性，还包括存储池和系统属性等，关于这部分内容，后面讲解 zap 时会详细提到。

那么对于 dnode_pyhs_t 结构体中，除了 union 中的字段，其他字段的作用是什么呢？下面来简单分享一下部分字段的含义，如表 2-2 所示。

表 2-2

字段	含义
dn_type	对应 dmu_object_type 中的内容
dn_indblkshift	这是间接块大小数值取对数的结果
dn_datablkszsec	文件的数据块和间接块大小，范围是从 512b 到 128kb
dn_checksum	数据校验的 checksum 类型，这里有多种，常见的如 sha256，具体的可以看 ZIO_CHECKSUM 这个枚举
dn_nlevel	记录 dnode 的 level，和前面提到的文件 level 0，1 和大于 1 的情况对应起来
dn_maxblkid	数据块的绝对 id 最大值
dn_compress	数据压缩的类型，具体见枚举 ZIO_COMPRESS 的数值

2.4 zfs 属性 zap

在 zfs 中，与属性相关的模块为 ZAP，其全称叫作 zfs attribute processor，是 DMU 中的一个子模块，这里提到了 attribute，与其属性值有关。在 Linux 中，也就是与 getattr，setattr 这些命令相关。同时 ZAP 对象可以用来保存数据集（dataset）的属性，用来查找文件系统中的对象，用来保存存储池的属性等等，它在 ZFS 整个系统中有十分重要的作用，如图 2-8 所示。

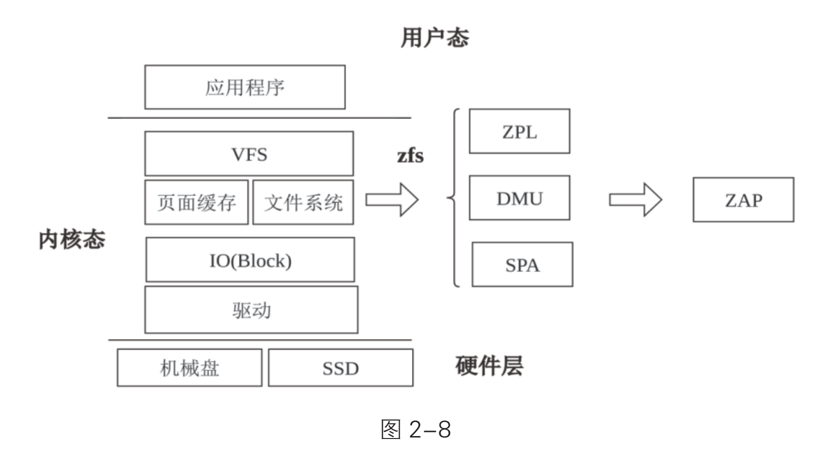


图 2-8

下面先来看一个简单的例子 .

```
1.  [root@gfsclient01 test-replica]# getfattr -n glusterfs.gfid.
    string  a.txt
2.  # file: a.txt
3.  glusterfs.gfid.string="71b9fb49-53ae-42f8-bb1b-b53af17521fc"
```

代码 2-8

所谓的属性，就是一个个键值对，这里可以设置很多，每一条都是一个属性值。除了 stat 命令看到的，还可以像上面这段代码一样设置一下。这里的例子，是 glusterfs 中经常会使用到的，而 glusterfs 则是一个分布式的文件系统，与 HDFS 和 CEPH 都是业界中比较有名的开源的分布式存储系统。glusterfs 使用扩展属性的目的之一，是为了记录文件的读写操作，简单来说就是当有一次写入的时候，就写入扩展属性并且递增相关的属性值，当操作完成后，开始递减，直到任务完成。通过这样简单的方式，来记录文件的操作过程。这个过程在 glusterfs 中称为 AFR。

zfs 中的 ZAP 有两种类型的对象，分别被称为 microzap 和 fatzap 属性。这二者的区别，这与前面提到的文件结构中的把属性内嵌进文件 header 中是非常相似的。而 fatzap 就是当属性多了以后，又或者添加一些需要很长的名称的属性，需要对其进行转换文件结构。当然 ZAP 中记录的不仅只是文件的属性信息，还有包括内部的一些属性信息，例如存储池的属性等。

2.4.1 zap

首先来看看 zap 的结构体，如下所示。

```
1.  typedef struct zap {
2.    dmu_buf_user_t zap_dbu;
3.    objset_t *zap_objset;
4.    uint64_t zap_object;
5.    struct dmu_buf *zap_dbuf;
6.    krwlock_t zap_rwlock;
7.    boolean_t zap_ismicro;
8.    int zap_normflags;
9.    uint64_t zap_salt;
10.
11.   union {
12.     struct {
13.       kmutex_t zap_num_entries_mtx;
14.       int zap_block_shift;
15.     } zap_fat;
16.     struct {
```

```
17.     int16_t zap_num_entries;
18.     int16_t zap_num_chunks;
19.     int16_t zap_alloc_next;
20.     avl_tree_t zap_avl;
21.    } zap_micro;
22.   } zap_u;
23.
24.  } zap_t;
```

代码 2-9

zap 中 zap_ismicro 就是用来判断是否为 microzap 类型的，下面的 union 联合体，则是用来区分二者不同类型的。这里需要留意一点，zfs 的代码在以前的版本（0.6.4）中，union 结构体中是各自保存了一个 block，也就是第一个数据块，对应 microzap 和 fatzap 二者的 header 结构，但是在 0.6.5 版本以后，删除了该代码，把内容移到了对应的 pyhs 结构中，如下所示。

```
1.  static inline zap_phys_t *
2.  zap_f_phys(zap_t *zap)
3.  {
4.   return (zap->zap_dbuf->db_data);
5.  }
6.
7.  static inline mzap_phys_t *
8.  zap_m_phys(zap_t *zap)
9.  {
10.  return (zap->zap_dbuf->db_data);
11. }
```

代码 2-10

阅读源码时，需要留意该结构体的变化，如果想了解更新更详细的改动，可以留意相关的 commit。[4]

zap_t 结构体初始化的代码可以看 zap_update 函数，可根据类型判断是否用 microzap 来分别调用不同的函数进行处理，如下所示。

```
1.  int
2.  zap_update(objset_t *os, uint64_t zapobj, const char *name,
3.     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx)
4.  {
5.   zap_t *zap;
6.   ...
7.
8.   if (!zap->zap_ismicro) {
9.    err = fzap_update(zn, integer_size, num_integers, val,
10.       FTAG, tx);
11.   zap = zn->zn_zap;
12.  } else if (integer_size != 8 || num_integers != 1 ||
13.      strlen(name) >= MZAP_NAME_LEN) {
14.   dprintf("upgrading obj %llu: intsz=%u numint=%llu name=%s\n",
15.       (u_longlong_t)zapobj, integer_size,
16.       (u_longlong_t)num_integers, name);
17.   err = mzap_upgrade(&zn->zn_zap, FTAG, tx, 0);
18.   if (err == 0) {
19.    err = fzap_update(zn, integer_size, num_integers,
20.        val, FTAG, tx);
21.   }
22.   zap = zn->zn_zap;
23.  } else {
24.   mzap_ent_t *mze = mze_find(zn);
25.   if (mze != NULL) {
26.    MZE_PHYS(zap, mze)->mze_value = *intval;
27.   } else {
28.    mzap_addent(zn, *intval);
29.   }
30.  }
31.  ...
32. }
```

代码 2-11

上述代码中 mzap_update 和 fzap_update 的函数分别对应 microzap 和 fatzap 结构体的初始化。

因此，先进一步了解二者不同类型结构体的区别，下面分别简单介绍一下。

2.4.2 microzap

microzap 实现了存取少量属性的一种简单方法。一个 microzap 对象只包含一个数据块，该数据块上存放 microzap 的条目（由 mzap_ent_phys 结构体表示），每个 microzap 条目结构存放一个属性（名字值对）。为了理解 microzap 和 fatzap 的区别，下面先来简单看看 microzap 的内容。

```
1. typedef struct mzap_phys {
2.  uint64_t mz_block_type;
3.  uint64_t mz_salt;
4.  uint64_t mz_normflags;
5.  uint64_t mz_pad[5];
6.  mzap_ent_phys_t mz_chunk[1];
7. } mzap_phys_t;
```

代码 2–12

mzap_phys 就是 microzap 的结构，其中 mz_block_type 就是 ZBT_MICRO，也就是 dmu 其中一个类型。这里 microzap 是使用了一个数据块来存放属性的，该数据块就是 mzap_phys 的字段 mz_chunk。

这里 mzap_phys 字段中的 mz_normflags 和 lookup 有关，也就是与遍历属性有关。这里有几个不同的设置值，可以参考 zap_micro.c 中的函数 mzap_create_impl 的注释内容，这里提到会和 zap_lookup_norm（）函数有关，感兴趣的可以自行追踪阅读代码。另外 mz_salt 字段是一个哈希值，用来识别不同的对象。

属性都是一个 key/value 对形式的，而 zfs 中对于属性的 key/value 长度都有限制，其最大长度分别由字段 ZAP_MAXNAMELEN 和 ZAP_MAXVALUELEN 来控制，其中单位是字节（byte），而默认的名称最大长度是 256 字节，属性值则是

1024*8 字节，也就是 8Mb。

mzap_phys 中存放属性的数据块是在 mz_chunk 中，因此下面来查看一下该结构体的内容。

```
1. #define MZAP_ENT_LEN  64
2. #define MZAP_NAME_LEN  (MZAP_ENT_LEN - 8 - 4 - 2)
3.
4. #define ZAP_NEED_CD  (-1U)
5.
6. typedef struct mzap_ent_phys {
7.  uint64_t mze_value;
8.  uint32_t mze_cd;
9.  uint16_t mze_pad; /* in case we want to chain them someday */
10.  char mze_name[MZAP_NAME_LEN];
11. } mzap_ent_phys_t;
```

代码 2–13

从这里可以看出 mze_name 就是记录属性中的名称的，而 mze_value 则是记录属性的数值的，也就是上面的例子中等号后面的内容。同时这里结构体中的数值大小，对应 mze_name 的大小是 50 字节（上面的 MZAP_ENT_LEN 的数值相减得到的），同时 mze_value 的大小是 64 字节。因此如果一个属性超过了这些限制，必然需要转换结构，也就是 fatzap 结构。mze_cd 则是用来寻找当 hash 相同时的属性，这里需要进行遍历，这里的实现和 hashmap 类似，关于该字段的使用，可以见如下代码所示。

```
1. static uint32_t
2. mze_find_unused_cd(zap_t *zap, uint64_t hash)
3. {
4.  mzap_ent_t mze_tofind;
5.  avl_index_t idx;
6.  avl_tree_t *avl = &zap->zap_m.zap_avl;
7.
8.  ASSERT(zap->zap_ismicro);
9.  ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
```

```
10.
11.  mze_tofind.mze_hash = hash;
12.  mze_tofind.mze_cd = 0;
13.
14.  uint32_t cd = 0;
15.  for (mzap_ent_t *mze = avl_find(avl, &mze_tofind, &idx);
16.         mze && mze->mze_hash == hash; mze = AVL_
     NEXT(avl, mze)) {
17.   if (mze->mze_cd != cd)
18.    break;
19.   cd++;
20.  }
21.
22.  return (cd);
23. }
```

代码 2-14

最后再来看一下，microzap 要增添属性时的函数调用，也就是 zap_t 结构体中，union 为 zap_micro 时对 avl tree 的操作，如下所示。

```
1.  static void
2.  mze_insert(zap_t *zap, int chunkid, uint64_t hash)
3.  {
4.   ASSERT(zap->zap_ismicro);
5.   ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
6.
7.   mzap_ent_t *mze = kmem_alloc(sizeof (mzap_ent_t), KM_SLEEP);
8.   mze->mze_chunkid = chunkid;
9.   mze->mze_hash = hash;
10.  mze->mze_cd = MZE_PHYS(zap, mze)->mze_cd;
11.  ASSERT(MZE_PHYS(zap, mze)->mze_name[0] != 0);
12.  avl_add(&zap->zap_m.zap_avl, mze);
13. }
```

代码 2-15

2.4.3 fatzap

下面来看一下 fatzap 的结构，fatzap 结构用一种灵活的方式来存储大量的属性，或者是用表示属性的名字值对有较长的名字或较长的值（不是 uint64_t 类型）的情况。

fatzap 和 microzap 不一样的地方在于，fatzap 有两个类型结构，分别是 ZBT_HEADER 和 ZBT_LEAF，根据名称可以看出，前者就是 fatzap 结构的 header 节点，而后者则是叶子节点。

fatzap 对象的第一个数据块上保存了一个 zap_phys_t 结构体，根据 hash 表的大小，会决定是否将 hash 表直接存放在 zap_phys_t 结构体中（hash 表要占用第一个数据块一半的空间。当索引表较大时，需要将它放在 zap_phys_t 之外）。下面来看一下 ZBT_HEADER 类型对应的结构体 zap_phys_t，如下所示。

```
1.  typedef struct zap_phys {
2.   uint64_t zap_block_type;
3.   uint64_t zap_magic;
4.
5.   struct zap_table_phys {
6.    uint64_t zt_blk;
7.    uint64_t zt_numblks;
8.    uint64_t zt_shift;
9.    uint64_t zt_nextblk;
10.   uint64_t zt_blks_copied;
11.  } zap_ptrtbl;
12.
13.  uint64_t zap_freeblk;
14.  uint64_t zap_num_leafs;
15.  uint64_t zap_num_entries;
16.  uint64_t zap_salt;
17.  uint64_t zap_normflags;
18.  uint64_t zap_flags;
19.
20. } zap_phys_t;
```

代码 2-16

fatzap 对象中所有的条目（key/value 数值对）都通过属性名的 64 位的 hash 值来记录。fatzap 用该 hash 值在一个 hash 表中进行索引，以得到存放属性名字值对应的数据块。hash 值中用来进行索引的比特数（prefix）由 hash 表的大小决定，hash 表会随着 fatzap 中存放的条目增多而变大。每一个 hash 表项指向一

个 fatzap 数据块，用 zap_leaf_phys 结构表示，每一个数据块又包含了很多个子块（chunk），用 zap_leaf_chunk 结构表示。每一个属性的名字和值存在于多个子块之中。

zap_table_phys 表示 hash 表的结构体。zt_blk，hash 表的第一个数据块号，当 hash 表不在 zap_phys_t 结构体内部时该字段有效，否则设为 0。zt_numblks 记录、hash 表占用的数据块数。当 hash 表不在 zap_phys_t 结构体内部时该字段有效，否则设为 0；zt_shift，hash 值中用来索引 hash 表的比特数。zt_nextblk 和 zt_blks_copied 在 hash 表改变大小时采用。

其结构关系如图 2–9 所示。

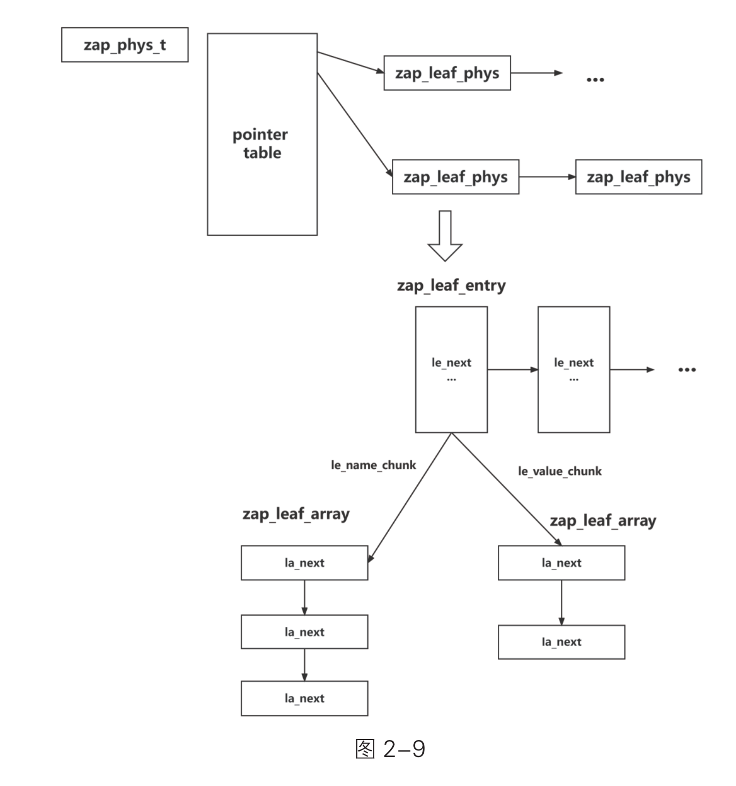


图 2–9

接着来看看 zap_leaf_phys_t 的结构，如下所示。

```
1.  typedef struct zap_leaf_phys {
2.   struct zap_leaf_header {
3.
4.    uint64_t lh_block_type;
5.    uint64_t lh_pad1;
6.    uint64_t lh_prefix;
7.    uint32_t lh_magic;
8.    uint16_t lh_nfree;
9.    uint16_t lh_nentries;
10.   uint16_t lh_prefix_len;
11.
12.
13.   uint16_t lh_freelist;
14.   uint8_t lh_flags;
15.   uint8_t lh_pad2[11];
16.  } l_hdr;
17.
18.  uint16_t l_hash[1];
19. } zap_leaf_phys_t;
```

代码 2–17

这里有很多结构的含义，与 microzap 中的类似，因此就不再过多重复介绍了。而以前的代码中，还会把 zap_leaf_chunk 的结构体放在 zap_leaf_pyhs_t 中，现在则独立出来，如下所示。

```
1.  typedef union zap_leaf_chunk {
2.   struct zap_leaf_entry {
3.    uint8_t le_type;
4.    uint8_t le_value_intlen;
5.    uint16_t le_next;
6.    uint16_t le_name_chunk;
7.    uint16_t le_name_numints;
8.    uint16_t le_value_chunk;
9.    uint16_t le_value_numints;
10.   uint32_t le_cd;
11.   uint64_t le_hash;
12.  } l_entry;
13.  struct zap_leaf_array {
```

```
14.     uint8_t la_type;
15.     uint8_t la_array[ZAP_LEAF_ARRAY_BYTES];
16.     uint16_t la_next;
17.   } l_array;
18.   struct zap_leaf_free {
19.     uint8_t lf_type;
20.     uint8_t lf_pad[ZAP_LEAF_ARRAY_BYTES];
21.     uint16_t lf_next;
22.   } l_free;
23. } zap_leaf_chunk_t;
```

代码 2–18

这里 zap_leaf_free 结构则是用来保存那些要释放的内容的，这里可以参考函数 zap_leaf_chunk_free 中的内容。另外当数据多了以后，pointer table 需要扩展，这里可以参考 zap_grow_ptrbl 函数。

2.5 位图与 metaslab

现在来了解一下 zfs 的 spa 模块，其中 metaslab 是一个非常有意思的模块，也是很重要的内容。从前面的内容可以知道 spa 负责存储池管理，而每个存储池可能会有多个不同的磁盘，不同磁盘之间可能是镜像，热备或者 Raid 关系等，这里我们先简化一下模块，以一个最基础的场景来先了解一下 spa。这里假设只有一个磁盘的话，那么在 windows 下，大家都知道要使用分区，也就是 c 盘，d 盘和 e 盘等，而这么多年的使用习惯下，大家都会默认把 c 盘作为系统盘来使用（这当然不是强制的，只是用户习惯形成的）。那么在 zfs 的磁盘管理单元中，metaslab 也可以理解为一个个分区，也就是对当前的磁盘进行划分不同的分区，并且每个分区负责一部分区域的空间管理。

那么这里 metaslab 的数量划分到底是怎样的呢？其如表 2–3 所示。

vdev size	metaslab count
< 8GB	~16
8GB~100GB	one per 512MB
100GB – 3TB	~200
3TB – 2PB	one per 16GB
> 2PB	~131,072

表格 2–3

这个表格的内容来源于 openzfs 的源码，在 vdev.c 文件中。由表 2–3 可以看到，大部分情况下，磁盘容量在 100GB ~ 3TB 的范围内，那么 metaslab 的数量大概就在 200 个左右，这也是很多文章所说的 200 的数字来源了。

在 zfs 的源码中，关于 metaslab 还有两个数值值得关注，分别是 ms_size 和 ms_count，这里数值关系如下所示。

```
1.  2^29 <= ms_size <= 2^34
2.  16 <= ms_count <= 131,072
```

代码 2–19

大家在这里可以思考一下，为什么要考虑对磁盘空间管理进行划分区域呢？这里有什么好处呢？如果整个磁盘都是由一个大的空间进行管理，那么并发问题会比较明显，同时空间碎片化管理的难度也会更大（因为每次做磁盘空间碎片化管理扫描的话，需要对当前整个空间管理记录进行扫描，那么如果一边读写一边修改，带来的并发处理问题会更加棘手，冲突概率也会增大）。除了这些，一般来说，磁盘空间使用，并不会是整个磁盘空间都会马上使用的，就像大家写入磁盘数据那样，通常是优先使用前面的一部分磁盘空间，然后写满了再到下一个区域。

同时，如果磁盘是机械硬盘的话，那么整个空间管理都是以磁盘为单位，其实也是不友好的，一个明显的特点是机械硬



盘是有柱面、扇区这些概念的，不同柱面的读写速度，切换不同柱面所带来的时间和性能成本不可忽略，因此在 Jeff 的博客中，还提到了考虑磁盘内外道扫描速度的差异，因此这些都可以成为考虑的因素之一。

当然，除了这些，还有一个问题大家可以考虑一下，不管是新的磁盘区域激活使用，还是有一个新的磁盘加入存储池进行扩容，新的磁盘空间加入，就一定会涉及数据均衡问题。也就是说，一旦有新的空间加入，那么对于新请求，权重该如何考虑呢？旧的数据文件需要进行迁移吗？新的文件创建，是不是应该偏向新的磁盘空间呢？这个问题，也不只是单机文件系统需要考虑的，分布式环境下一样需要考虑，例如 glusterfs 的 volume 扩容之后，数据均衡都是要考虑的。

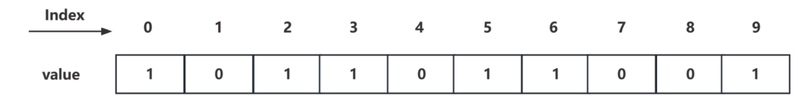
对于数据均衡问题，笔者认为，这不是技术问题，更多偏向于业务问题，因为不同场景下需求不同。有些企业如果是存的视频数据，尤其是一些监控视频、历史监控数据等，这些数据内容都带有明显的时间维度。这时候数据倾斜是正常的，因为旧数据存放在旧磁盘空间上，而且旧数据的读取次数会逐渐减少，甚至成为冷数据和无效数据（所谓无效，也是根据业务来考虑的）。所以这些问题的考虑，更多要结合业务场景来思考。

同时，这里也不能把太多新的文件读写创建请求都放在新的磁盘空间中，这样做也会导致新的磁盘空间被大量使用，如果是单独的磁盘的话，该节点的性能瓶颈短时间内都会集中在该磁盘上，这样做也是非常不友好的。因此为了均衡以上提到的问题，在分布式环境下，就需要做 QoS 了，核心需求之一就是分布式环境下的流量负载均衡。

在单机环境下，如果有多个磁盘可以选择，那么还要考虑不同磁盘 IOPS 的不同（通常建议生产环境下，配置都是标准化和统一的，也就是不会出现 cpu 核数，磁盘容量差异很大的情况，或者有的节点有固态，有的没有，那么分布式多副本情况下，对读写影响是比较大的）。如果是有固态和机械硬盘，还要考虑二者的性能差异。

了解了 metaslab 的数量划分情况，那么，到底在每个 metaslab 中是如何做空间管理的呢？在深入了解 metaslab 之前，先来分享一下以前的磁盘空间管理技术——位图 BitMap。

位图技术的使用，在早期的文件系统如 ext2 中是有应用的，其实原理很简单，就是根据格式化时，指定磁盘数据的块大小，然后计算出所需要的位图空间大小。以默认的 4k 为例，位图就是用一个比特位，0 和 1 来表示当前的 4k 空间是否被使用过。当然在 ext 文件系统中，磁盘空间划分叫作 block group。因此每个 block group 中都有一个对应的位图来记录这些数据，如图 2-10 所示。



Bitmap

图 2-10

使用位图来做空间管理，好处为简单和一目了然。只要空间被使用了，那么该比特位置为 1 即可。这个数据的使用在 ext2 文件系统如图 2-11 所示。

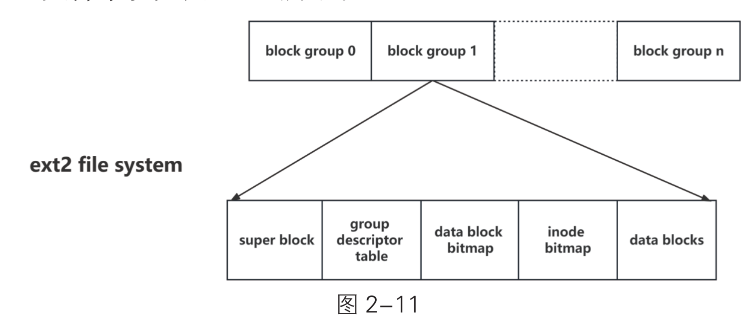


图 2-11

这里使用位图会出现一些麻烦。首先就是空间问题，一个几太字节的空间数据则可能需要几百兆的位图数据空间，那么随着数据的增长，现在动辄几百太字节甚至更多的数据存储，那么文件的位图数据大小就不可忽视了。同时申请和释放都要持久化位图的结构到磁盘上，虽然可以只持久化修改的 block group 下的位图，但是如果文件跨了很多 block group 的话，那么更新的 IO 操作也非常的难。

对一个 1GB 的文件系统来说，如果它的 Bitmap 是 32KB，很容易将这部分数据存储到内存中，也很方便对其进行扫描，找到空闲空间。那么对于一个 1TB 的文件系统来说，Bitmap 是 32MB，同样可以很方便地存放在内存中，但是扫描这 32MB 数据的每一位就不是那么容易的事情了。而对于 1PB 的文件系统来说，它的 Bitmap 是 32GB，这对于大多数机器的内存来说都是不能接受的。这就意味着要从硬盘上将这 32GB 的数据读出，然后扫描每一位数据，这将会更慢。很显然，这种方式并不适合。

一个优化方法是将 Bitmap 分成许多个小块，每次只管理一个小块中的位。比如，对于使用 4KB 块大小的 1PB 文件系统来说，空闲空间可以划分为一百万个小的 bitmap，每个大小为 32KB。同时将概要信息（使用一百万个整数来表示每个 bitmap 中的空间）存放在内存中，这样一来就可以很容易找到空闲空间，而且扫描小的 bitmap 要高效得多。

但是这里仍然有一个很严重的问题，Bitmap（s）不仅是在分配新 block 的时候需要被更新，在 block 被释放的时候也要被更新。文件系统控制着分配的位置（这决定数据将被写入哪些 block），同时也要控制释放的位置。一些很简单的操作，比如“rm -f ”会造成整个设备上的 block 被释放。以 1PB 的文件系统为例，在最坏的情况下，删除一个 4GB 的数据（一百万个 4K 的 block）将需要对这百万个 bitmaps 进行读、修改、写回操作。

也就是说，删除一个 4GB 的文件可能将涉及 2000 万次磁盘的 I/O 操作。

因此 zfs 并没有使用位图的方式来管理磁盘空间，那么这里 metaslab 是如何做的呢？如图 2–12 所示。

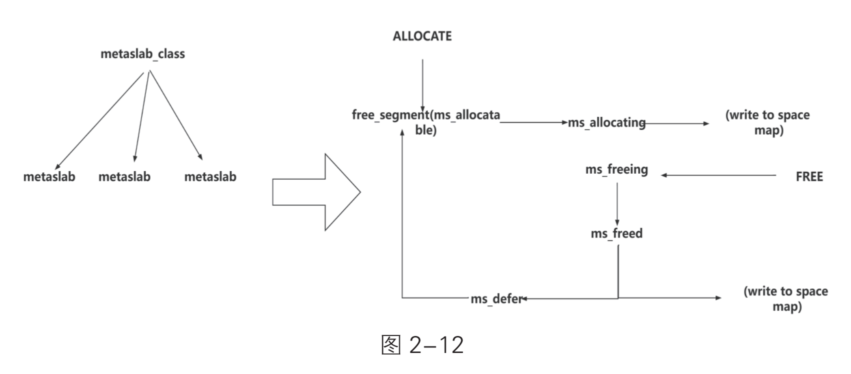


图 2–12

那么当 block 准备被释放的时候，会被添加到 ms_freeing 中，再添加到 ms_freed 中。

根据 zfs 对 ms_freeing 和 ms_freed 的数据进行结构注释，其中 ms_freeing 是正在处理的 txg 事务要释放的空间，而 ms_freed 是当前 txg 事务已经释放的空间（txg 是 transaction group，事务组，这里可以先简单理解为每个事务的处理即可，后面会提到）。

那么当 txg 已经处理完成之后，并且空间释放了，block 空间会被转移到 ms_defer tree 中，这里可以称为延迟释放树。延迟释放的概念是因为虽然当前的 block 空间已经被释放了，但是并不能马上被重新利用，需要等到一段时间之后才能重新使用。

那么这个一段时间是多久呢？这里就是参数 TXG_DEFER_SIZE 的作用了，这里是默认为 2，意味着就是在后续两个 txg 执行完成后才能重新使用。

举个例子，如果当前 txg 是 1，释放了 block 空间，那么只有 1+ TXG_DEFER_SIZE，也就是当执行到 txg 为 3 的时候，才能重新利用。

这是为了数据安全，因为如果要释放的数据马上可以被其他新的请求使用，一旦事务执行失败了，那么旧的数据空间可能已经被覆盖无法进行回滚，会造成数据丢失、内存踩踏等现象，这是非常危险的行为。

最后来看一下 metaslab 的结构。

```
1.  struct metaslab {
2.          space_map_t   *ms_sm;
3.          uint64_t ms_id;
4.          uint64_t ms_start;
5.          uint64_t ms_size;
6.          uint64_t ms_fragmentation;
7.
8.          range_tree_t *ms_allocating[TXG_SIZE];
9.          range_tree_t *ms_allocatable;
10.         range_tree_t *ms_freeing;
11.         range_tree_t *ms_freed;
12.         range_tree_t *ms_defer[TXG_DEFER_SIZE];
13.           ...
14. }
```

代码 2-20

metaslab 中使用的 range_tree 其实是 avl tree。这里有一个字段 ms_sm，这个就是 spacemap，也就是与日志相关的，每个 metaslab 下面会关联很多日志信息，这些日志都会记录当前的 metaslab 申请和释放的空间，关于这部分内容见下一个小节。

2.6 zfs SpaceMap 和 MetaSlab

前面提到，每个 metaslab 字段中都会关联 spacemap，那么既然涉及日志，必然会有一个问题需要考虑，那就是日志什么时候过期，又或者说日志什么时候被删除？

对于 spacemap 日志来说，因为记录的信息，也就是 metaslab

的 tree 是在内存中的，理论上说，累积了一段时间之后，如果日志数量很多再落盘，那么在这个间隔时间内，若节点出现问题了，需要把日志重做，也就是重新解析，再构建好 metaslab 的 tree 信息，这个时间会很长，同时因为 metaslab 的数量还与磁盘大小有关，通常是 200 个。因此一旦日志累积过多，那么可能需要重做日志的 metaslab 也不少。

当然这里还有一个问题需要考虑，那就是日志的数量是否应该要限制呢？因为当日志数据量很大了以后，占用的空间是不小的，如果一个系统开启了 debug 又或者 trace 级别的日志，日志刷新的频率是非常高的。（这一点 glusterfs 的 volume 就可以动态修改日志级别进行调试查看问题）。这里日志的数量多到一定程度时，也需要触发 metaslab 信息落盘的操作，来进一步释放日志空间。那么这里设置上限的时候，有两种方式可以对其考虑，一是考虑绝对值，二是考虑百分比。

对于一个磁盘来说，百分比很好理解，例如限制 spacemap 日志信息为 1G，那么达到阈值的 70% 甚至 50% 以后，就要考虑进行刷新，尽量把使用量减少到一半以下，用来预防部分大 IO 流量时刻，在短时间内日志信息很多，一方面正常文件的读写请求很多，同时日志信息也不少，但是 zfs 来不及处理日志删除等任务（对于磁盘空间也是相似的，当磁盘空间使用量达到一半以上时，就要开始留意了，如果达到 70% 以上，就属于一个比较谨慎的时候了，一旦磁盘被使用完，或者磁盘使用量达到 90% 以上，很容易出现各种问题，如性能快速下降）。

那么绝对值又是如何考虑的呢？当达到规定数值就进行数据落盘。因为磁盘大小是不一样的，有些是以 GB 为单位的，常见的如 512GB、256GB 的固态等，但是也有一些是以 TB 为单位的，尤其是机械硬盘，现在上太字节的机械硬盘是很常见的。在生产环境中，笔者使用过 12TB 的机械硬盘，当时节点是 12*12TB

的。若这时候还是以百分比来考虑的话，就会显得有点大了，因此也有考虑绝对值的，例如 1G 的磁盘容量大小。

当两个数值放在一起的时候，该如何考虑呢？这种情况，一般是采用更加保守的方式，也就是取二者的较小值。

考虑了 spacemap 日志的大小之后，下面先来看看 spacemap 中记录了什么内容。一些空间常见的申请和释放过程如图 2-13 所示。

图 2-13

spacemap 中主要是记录了 metaslab 的日志释放和申请，也就是源码中的 maptype_t 字段的作用 .

```
1. typedef enum {
2.  SM_ALLOC,
3.  SM_FREE
4. } maptype_t;
```

代码 2-21

刷新的时候，也就是事务组执行的时候，把 metaslab 的信息落盘，这里有两个字段就是与 spacemap 的申请释放日志相关的，如下所示。


```
1. struct metaslab {
2.     range_tree_t *ms_unflushed_allocs;
3.     range_tree_t *ms_unflushed_frees;
4.     ...
5. }
```

代码 2-22

每次刷新的时候，该释放多少日志空间，又或者说，该选择多少个 metaslab 来进行考虑刷新落盘信息呢？因为通常每个磁盘大部分情况下大约 200 个 metaslab（这个数值没印象可以看前面小节内容）。那么每次都需要考虑把全部的 metaslab 刷新落盘吗？这当然不需要，前面提到一个日志的限制数值，也就是说如果目前日志数量信息远远没有达到限制阈值，那么每次事务组执行事务都会持久化 metaslab 的信息。因为 metaslab 的信息也不是每个都会均匀使用的，可能有些 metaslab 累积的信息多，也有些很少。因此 openzfs 曾经考虑过使用预测的方式来释放日志空间。所谓预测，就是根据考虑前几次的 metaslab 落盘时释放的日志数据量，得到一个平均值，再与日志数据量阈值进行综合考虑。

另外关于日志，这里还需要留意一个问题，因为 spacemap 中记录的都是 metaslab 的申请和释放信息，这也就是会出现一些情况：spacemap 中的日志很多，但是实际上，都是相同或者有重叠的区域。例如一个区域 [1，10] 被申请了空间，若中间又被释放和再次申请等，也会产生很多 log 日志，如果这些日志累积的信息过多了，落盘的时候也会无形中增加了 IO 负担，如图 2-14 所示。

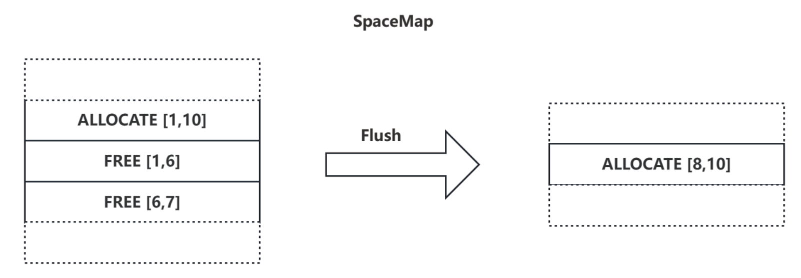
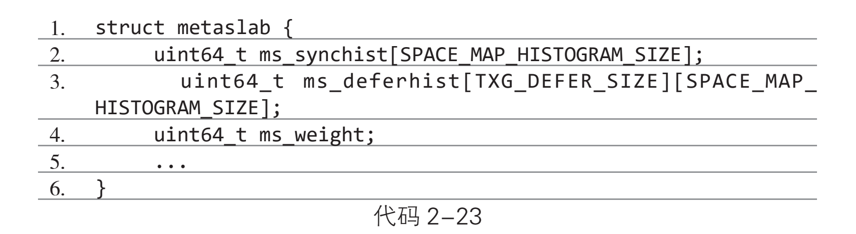


图 2-14

这里就引申出一个概念，叫作碎片化，在了解这个概念之前来先看一下 metaslab weight。当请求写入磁盘的时候，需要选择一个 metaslab 来进行考虑，那么这个权衡的字段是 ms_weight，如下所示。



```
1. struct metaslab {
2.       uint64_t ms_synchist[SPACE_MAP_HISTOGRAM_SIZE];
3.          uint64_t  ms_deferhist[TXG_DEFER_SIZE][SPACE_MAP_
   HISTOGRAM_SIZE];
4.       uint64_t ms_weight;
5.       ...
6. }
```

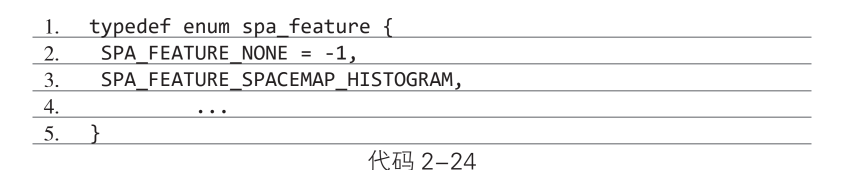
代码 2-23

metaslab 中的字段 ms_synchist 相当于 ms_freeing + ms_freed，而 ms_deferhist 每个数组则相当于对应的 ms_defer。

这里 ms_weight 的考虑就会和碎片化有关了，所谓碎片化，这里 zfs 会使用数组来记录一下不同的大小区间申请释放的情况（这里 zfs 的说法是 histogram，直方图，其实可以简单理解为不同区间的申请数据，放在表格上就类似一张直方图那样）。

对于数据的空间申请，这里的数据块大小是有不同区间的，从 512B 到 16MB，其会记录不同区间大小的数据数量，这些区间见 zfs_frag_table 字段，有 4k、16k 这些常见的大小，也有 1M 和 16M，这些数据数量汇总起来，就是可以形成一个直方图那

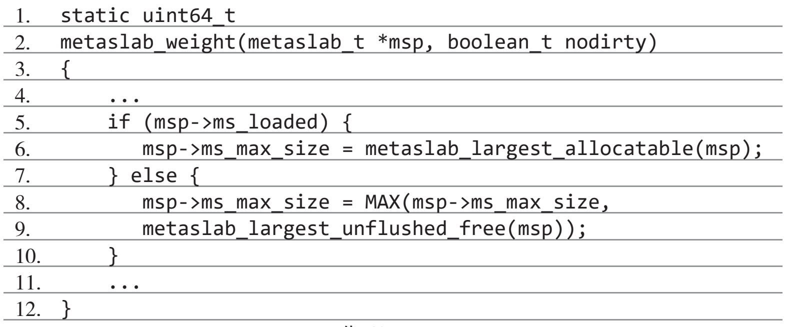
样的统计输出了，这就是 histogram。当然直方图的功能，需要开启一个 Feature，如果该 Feature 不开启，其计算方式是不同的。该 Feature 是 SPA_FEATURE_SPACEMAP_HISTOGRAM（见 spa_feature 枚举中的内容）。



```
1. typedef enum spa_feature {
2.  SPA_FEATURE_NONE = -1,
3.  SPA_FEATURE_SPACEMAP_HISTOGRAM,
4.          ...
5. }
```

代码 2-24

同时关于 metaslab 的权重统计，见 metaslab_weight 函数，如下所示。



```
1. static uint64_t
2. metaslab_weight(metaslab_t *msp, boolean_t nodirty)
3. {
4.     ...
5.     if (msp->ms_loaded) {
6.        msp->ms_max_size = metaslab_largest_allocatable(msp);
7.     } else {
8.        msp->ms_max_size = MAX(msp->ms_max_size,
9.        metaslab_largest_unflushed_free(msp));
10.    }
11.    ...
12. }
```

代码 2-25

从这段代码中可以看出，如果是已经被使用加载的 metaslab，那么这里考虑其最大的可用空间。对于权重的选择，一般主要有两种方式，一是考虑全部的可用空间，二是考虑最大的连续可用空间。这两种方式都有优缺点。zfs 这里选择了最大的可用空间。其可用空间需要考虑到 metaslab 的延迟释放问题，也就是说，当一个空间确定要被释放掉以后，其是会存放

在 deferred tree 中的，同时还会保存在前面提到的 unflushed tree 中。因此这里的实际可用空间是需要考虑延迟释放部分的空间的。这也就是 metaslab_largest_unflushed_free 函数中所做的事情。

```
# zpool create hilton raidz2 vdc vdd vde vdf vdg spare vdh

# zpool status
  pool: hilton
 state: ONLINE
  scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        hilton      ONLINE       0     0     0
          raidz2-0  ONLINE       0     0     0
            vdc     ONLINE       0     0     0
            vdd     ONLINE       0     0     0
            vde     ONLINE       0     0     0
            vdf     ONLINE       0     0     0
            vdg     ONLINE       0     0     0
        spares
          vdh       AVAIL

errors: No known data errors

# mkdir -p /mnt/zpool/hilton

# zfs set mountpoint=/mnt/zpool/hilton hilton

# df -h /mnt/zpool/hilton/
Filesystem      Size  Used Avail Use% Mounted on
hilton           86G   17M   86G   1% /mnt/zpool/hilton
```

代码 2-27

这里创建了一个 raidz2 的 zpool，每个磁盘都是 30G 的大小，名为 hilton，接下来创建挂载点并且导入一些数据，下面来继续看看 metaslab 的信息。

```
# zdb -m hilton > /tmp/metaslab.txt
# cat /tmp/metaslab.txt


Metaslabs:
 vdev          0
 metaslabs   149   offset                spacemap          free
 ---------------  -------------------  ---------------  ------------
 metaslab      0   offset      0   spacemap     81   free      996M
space map object 81:
  smp_length = 0x568
  smp_alloc = 0x1bc6000
 metaslab      1   offset   40000000   spacemap     80   free    1022M
space map object 80:
  smp_length = 0x598
  smp_alloc = 0x196800
 metaslab      2   offset   80000000   spacemap     79   free     895M
space map object 79:
  smp_length = 0x498
  smp_alloc = 0x817f200
 metaslab      3   offset   c0000000   spacemap     78   free    1024M
space map object 78:
  smp_length = 0x440
  smp_alloc = 0x49e00
 metaslab      4   offset  100000000   spacemap     77   free     995M
space map object 77:
  smp_length = 0x488
  smp_alloc = 0x1d6ca00
 metaslab      5   offset  140000000   spacemap     76   free    1024M
space map object 76:
  smp_length = 0x460
  smp_alloc = 0x45000
 metaslab      6   offset  180000000   spacemap     75   free     995M
space map object 75:
  smp_length = 0x508
  smp_alloc = 0x1d1ae00
 metaslab      7   offset  1c0000000   spacemap     74   free    1024M
space map object 74:
  smp_length = 0x4e8
  smp_alloc = 0x4fe00
 metaslab      8   offset  200000000   spacemap     73   free    1024M
space map object 73:
  smp_length = 0x578
  smp_alloc = 0x1c800
 metaslab      9   offset  240000000   spacemap      0   free       1G
 metaslab     10   offset  280000000   spacemap      0   free       1G
 metaslab     11   offset  2c0000000   spacemap      0   free       1G
```

```
48.  metaslab    12  offset   300000000  spacemap   0  free   1G
49.  metaslab    13  offset   340000000  spacemap   0  free   1G
50.  ...
51.  metaslab   146  offset  2480000000  spacemap   0  free   1G
52.  metaslab   147  offset  24c0000000  spacemap   0  free   1G
53.  metaslab   148  offset  2500000000  spacemap   0  free   1G
```

代码 2-28

通过 zdb，这里可以看到 metaslab 有 148 个，并不是所有的 metaslab 都在使用，当数据越来越多的时候，后面的 metaslab 才会有数据写入。

```
1.  ~# zdb -h  hilton
2.
3.  History:
4.  2022-08-07.14:04:06 zpool create hilton raidz2 vdc vdd vde vdf vdg spare vdh
5.  2022-08-07.14:09:56 zfs set mountpoint=/mnt/zpool/hilton hilton
```

代码 2-29

这里还可以使用 zdb -h 命令来查看 zpool 相关的操作记录。

```
1.  ~# zdb -mmm  hilton
2.
3.  Metaslabs:
4.          vdev          0
5.          metaslabs   149   offset          spacemap        free
6.          ---------------  ------------------  ---------------  ------------
7.          metaslab    0   offset      0   spacemap     81   free     996M
8.                        segments     17   maxsize    996M   freepct   97%
9.          In-memory histogram:
10.                                12:       6 ******
11.                                13:       3 ***
12.                                14:       4 ****
13.                                15:       2 **
14.                                16:       1 *
15.                                17:       0
16.                                18:       0
17.                                19:       0
18.                                 ...
19.                                29:       1 *
20.          On-disk histogram:                 fragmentation 0
21.                                11:       1 *
22.                                12:       7 *******
23.                                13:       4 ****
24.                                14:       6 ******
25.                                15:       2 **
26.                                16:       0
27.                                17:       0
28.                                ...
29.                                29:       1 *
30.          ...
```

代码 2-30

这里可以使用 zdb -mm 来输出更加详细的信息，其还包括了与碎片化相关的内容，因为 zfs 的 block size 分配是动态的，这里提到的 In-memory histogram 中的数字，如 12 则是 2 的 12 次方，也就是 4096，也就是 4KB 大小，这里有 6 次分配。

这里需要注意的是 spacemap 的大小默认是 4KB 大小（早期版本，目前已经更改为 128KB，为了适配大磁盘，该大小不影响理解），前面提到的不同区间大小，其是用户数据文件大小，而 spacemap 是内部空间管理使用的，因此二者是不同的。

4KB 大小的 spacemap，在较大的磁盘中，如 10TB 的磁盘，如果频繁有读写申请，那么 spacemap 记录有很多，同时对每个 4KB 的内容进行读取，也需要耗费很多时间。因此为了减少 spacemap 的数量，对 metaslab 的大小做了调整，通过减少 metaslab 的数量，可以进一步减少 spacemap 的数量。也就是上面看到的，每个 metaslab 管理的区间为 1GB，更早版本的 zfs 中 metaslab 的管理区间大小是以 MB 为单位的。

2.7 zfs 校验码

为了保护数据的完整性，每次在写入磁盘之前，都会对数据进行校验，而这个校验的结果就是 checksum，其使用的算法可以有很多，常见的有 sha256 和 md5，而这些数据块校验结果的存放，zfs 有一个独特的设计，会把结果存放在指向该数据块的间接块里，如图 2-15 所示。

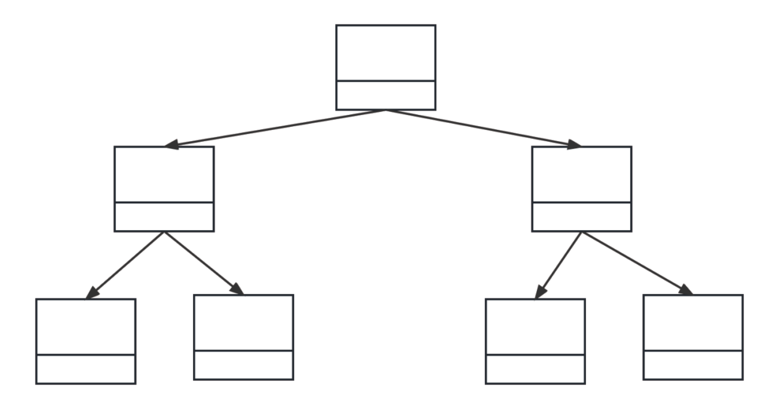


图 2-15

zfs 的独特设计就是会把 checksum 的计算结果存放在指向该叶子节点的间接节点上，而根节点的计算结果则存放在自身上面。这种设计的好处是什么呢？首先如果不存放在这里的话，那么其他的文件系统是如何设计的呢？有些会专门弄一个 checksum file 单独存放，如果要校验，则会进行相对多一次的访问了，并且重新生成一个新的文件，该文件的管理也是一个新的问题。而 zfs 的这种巧妙设计的好处为，除了减少了一个文件，在文件自修复的过程中也是有非常大的作用的在一些文件系统里面，还有一个叫作 merkle tree 的概念，merkle tree 的出现，是在网络传输的过程中，会把一个数据切成多个小的数据块再进

行传输，那么就需要计算这些不同的小块的 checksum 了，再得到一个最终的整体文件的 checksum。

这个概念和文件系统有什么关联呢？在文件系统里面，叶子节点就是数据块，正常的一个文件，如何快速知道是否有损坏呢？如果是镜像存储池，如何快速对比两个文件是否相同呢？这里如果全部计算叶子节点的 checksum 的话，速度就比较慢了，一个比较有效率的方式是对间接节点指向的叶子节点的 checksum 进行再次计算 checksum，得到一个叫作 merkle checksum 的结果，这样的话，因此要对比文件是否相同，先比较根节点是否相同，不相同再对比下面的间接节点，这样一层层对比，就可以减少很多的计算了，速度也快很多。

对于其他的文件系统来说，例如 btrfs 中，chcksum 是有一颗单独的树来进行保存结果的，叫作 checksum tree，该设计和 zfs 是不同的，大家感兴趣的话可以对比着学习二者设计的不同。[3]

2.8 zfs 缓存概念理解

前面提到了 zfs 存储池 / 空间管理 Metaslab 这些内容，现在用一张详细的 zfs 架构图来理解和认识一下 ZFS 的不同模块和整体架构图特点，如图 2-16 所示。

对于 zfs 来说，对外是提供 ZPL 和 ZVOL 的，前者就和正常文件系统使用一样，在 zfs 中可以创建一个 dataset（数据集）进行挂载使用，而 zvol 则是一个块设备，可以使用 iscsi 提供给远端的程序使用。在生产环境中，k8s 的后端 pvc 就可以使用 zfs 提供的 zvol 块设备进行数据落盘持久化。

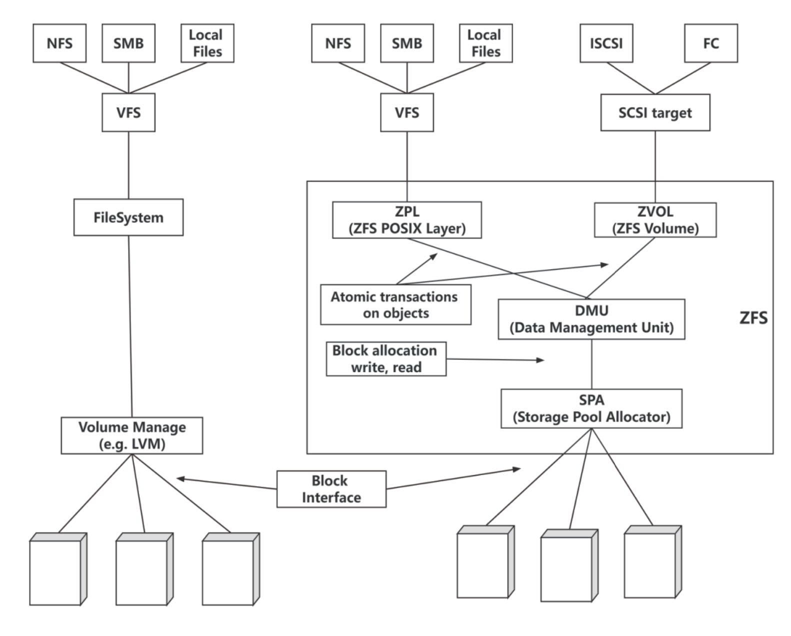


图 2-16 zfs 架构图

2.8.1 ARC 和 L2ARC

在文件系统中，不管是 zfs 还是 ext4，都要经过 VFS，再进入各自的文件系统中处理，那么在读写数据的时候，有一个重要的东西叫作 page cache，这个是 Linux 系统的缓存。而在 zfs 中有一个类似的模块叫作 ARC 和 L2ARC。不管是 page cache 还是 ARC，本质上都是 LRU 队列，也就是先进先出队列。ARC 的英文是 Adaptive Replacement Cache，自动调节的可替代的缓存。

有了 page cache 之后，当需要从磁盘读取数据的时候，可以把数据放在缓存中，也就是内存里面，这样可以加速读取文件的速度，尤其是文件需要预读的时候，也就是当打开一个新的文件，文件系统可能会根据需要读取文件的区间，把后面的内

容也提前加载一下，这样可加快下一次的读取，文件预读的策略对于大文件的读取是非常有效的。

zfs 的 ARC 设计是两个队列，分别是数据最近使用队列（MRU）和最近频繁使用队列（MFU）。这里为什么需要两个队列呢？可以考虑一个场景，当晚上或者某个时间内需要对数据做备份时，这时候需要进行全盘扫描或者扫描存储池内大量的文件然后加载到缓存中，接着发送给备份节点或者第三方的系统。如果只有一个队列，也就是 MRU，那么在扫描时，大量的文件数据，会冲刷掉其他正常读写的数据，会造成缓存丢失，造成其他正常读写请求的性能大幅度下降且不稳定，这种情况被称为缓存置换（ cache thrashing）。因此为了避免这种情况出现，对于那些在缓存中需要多次读写的数据，会加载到 MFU 队列中，这时候后台的定时扫描备份任务，也不会影响到正常的任务执行。

既然涉及缓存，其必然会有调整缓存大小的相关参数，这里比较常用的就是设置最大和最小值了。对于小于 4GB 的系统，这里一般是不建议进行调整的，因为内存太小了。而大于 4GB 的，默认是 1GB 大小的缓存。当然具体的参数可以看配置文件数值。

```
1.  # cat /proc/spl/kstat/zfs/arcstats |grep c_
2.  c_min                          4    128832000
3.  c_max                          4    2061312000
4.  arc_no_grow                    4    0
5.  arc_tempreserve                4    0
6.  arc_loaned_bytes               4    0
7.  arc_prune                      4    0
8.  arc_meta_used                  4    1228584
9.  arc_meta_limit                 4    1545984000
10. arc_dnode_limit                4    154598400
11. arc_meta_max                   4    4263256
```

```
12. arc_meta_min                    4    16777216
13. async_upgrade_sync              4    0
14. arc_need_free                   4    0
15. arc_sys_free                    4    64416000
16. arc_raw_size                    4    0
```

代码 2-31

上面显示的 c_min 和 c_max 的数值是以 byte 为单位的，如果想要修改对应的大小，可以修改配置文件 /etc/modprobe.d/zfs.conf 的配置。

前面提到了 ARC 是用于缓存文件数据的，那么其缓存文件的什么数据呢？是文件元数据还是读写数据呢？这里也是可以进行调整的，在创建数据集时，指定只缓存文件的元数据信息，如下所示。

```
1.  # zpool status
2.    pool: hilton_raidz2
3.   state: ONLINE
4.  config:
5.
6.          NAME             STATE     READ WRITE CKSUM
7.          hilton_raidz2    ONLINE       0     0     0
8.            raidz2-0       ONLINE       0     0     0
9.              vdc          ONLINE       0     0     0
10.             vdd          ONLINE       0     0     0
11.             vde          ONLINE       0     0     0
12.
13. errors: No known data errors
14.
15. # mkdir -p /mnt/hilton_raidz2
16.
17. # zfs set mountpoint=/mnt/hilton_raidz2 hilton_raidz2
18.
19. # zfs create -o primarycache=metadata hilton_raidz2/test1
20. # zfs list
21. NAME                  USED  AVAIL     REFER  MOUNTPOINT
22. hilton_raidz2         136K  28.8G    23.9K  /mnt/hilton_raidz2
23.
24. hilton_raidz2/test1  23.9K  28.8G    23.9K  /mnt/hilton_raidz2/test1
25.
26. # zfs get primarycache hilton_raidz2/test1
27. NAME                 PROPERTY      VALUE        SOURCE
28. hilton_raidz2/test1  primarycache  metadata     local
```

代码 2-32

什么场景下可以考虑使用只缓存文件元数据的策略呢？对于一些冷数据的节点，读取数据的概率非常小，因此没有必要把数据读取到缓存中，但是可能需要对读取文件的元数据进行校验，同时也可以加快 ls 等命令操作的速度。当然，只缓存元数据还可以缓存非常多的文件元数据。文件元数据的大小是固定的，再根据缓存大小，就可以大概计算出可以缓存的文件元数据数量。

zfs 的缓存里面，除了 ARC，还有一个东西叫作 L2ARC。L2ARC 的出现其实也是为了解决当 ARC 缓存数据太多，导致一部分缓存数据需要被置换落盘时，找到一个地方来放置这些“溢出”的缓存，这个原理和 swap 是类似的。因此从这里可以知道，只有被 ARC 缓存过的内容，才会被置换到 L2ARC 中，所以前面的 ARC 设置了只缓存元数据信息，L2ARC 虽然按照默认缓存所有的内容，但是实际上并不会有其他的内容会置换到 L2ARC 中的，如下所示的设置。

```
1. ~# zfs get primarycache,secondarycache hilton_raidz2/test1
2. NAME                 PROPERTY        VALUE        SOURCE
3. hilton_raidz2/test1  primarycache    metadata     local
4. hilton_raidz2/test1  secondarycache  all          default
```

代码 2-33

这里还有一个小的知识点值得留意一下，一般来说，ARC 的设计不只是在文件系统中有，在数据库中也是存在的，而且

也是两个队列的设计，那么对于两个队列的容量比例的划分，通常有对半均分，也有用 3∶7 来进行拆分的，而 zfs 在这一点上，相对灵活一点，可以根据工作负载来微调比例，只要没有达到其上下限，那么 MRU 和 MFU 队列的大小是弹性的，当然一开始默认比例也是对半均分的。当然如果系统内存使用很紧张的时候，那么缓存的容量也是会减少的。同时在 MRU 和 MFU 中，都有一个 ghost 列表，该列表主要就是记录各自队列缓存要淘汰的页面信息。

2.9 zfs 缓存实现

前面提到了 arc 和 l2arc 的内容，但是还有一些内容是值得关注的，如果缓存数据被替换到了 l2arc 中，且 arc 里面并没有的话，那么读取数据的时候，该如何知道缓存是否在 l2arc 呢？还有 zfs 为了优化内存的使用率，做了一些怎样的优化呢？这些内容也是本小节关注的点。

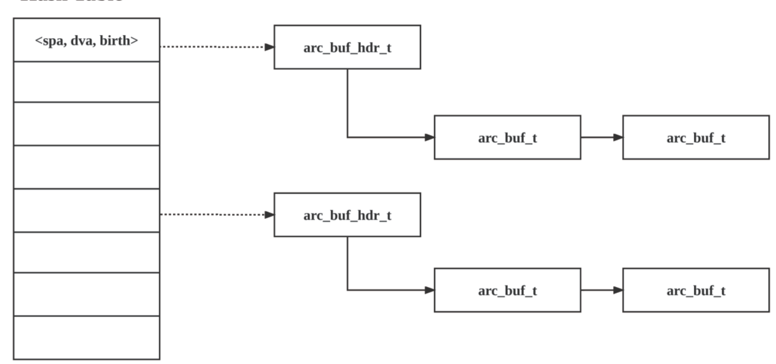
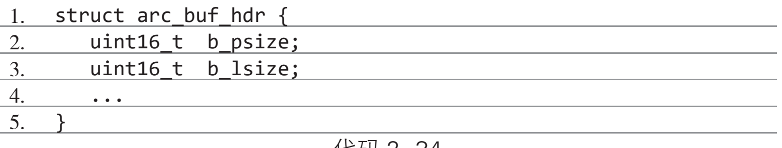


图 2–17

当要读取数据缓存的时候，需要先申请一个缓存的头部信息，也就是 arc_buf_hdr_t 结构体，该结构体的信息会和 Hash Table 关联，其中主要以 spa，dva 和 birth time 来进行区分，其中 birth time 是该缓存数据的 txg 信息。在后面有读的请求过来的时候，也是会查询哈希表中是否有数据。

而每个 arc_buf_hdr_t 结构后面是关联着 arc_buf_t 结构，这就是直接指向数据的结构体，每个 arc_buf_t 可以看成一份数据，以此来组成一个链式表。同时可以从图 2–17 中看到，每个 arc 头部结构体可能会指向多个 arc buffer 结构（arc_buf_t），这是有可能因为读取的数据是来源于不同的 dataset，例如快照或者克隆的数据。

同时在 zfs 中，为了节省内存的使用率，一般来说在不同的操作系统中，都会有缓存压缩的功能，通常的做法则是把要淘汰出缓存的数据进行压缩，然后放到 swap 中交换出去。这里就有一个问题了，压缩后的数据大小可能是会改变的（有时候不一定会改变，也就是压缩并没有起作用，对于一些视频，压缩效果不明显）。这里就有两个字段就可以起到作用了，也就是会用两个字段来分别记录压缩前后的数据大小。当然 zfs 的 arc 压缩功能开启，可以通过设置字段 zfs_compressed_arc_enabled 来进行控制。



```
1. struct arc_buf_hdr {
2.     uint16_t  b_psize;
3.     uint16_t  b_lsize;
4.     ...
5. }
```

代码 2–34

如果这个时候数据被转移到了 l2arc 中，并且 arc 中的缓存数据被清理了，那么该如何知道该数据是否在 l2arc 呢？在 arc_buf_hdr_t 结构体中有一个字段 l2arc_buf_hdr_t 则是用于此地。

zfs 的 arc 几种状态，也就 arc_state_type_t 的字段所示。

```
1. typedef enum arc_state_type {
2.   ARC_STATE_ANON,
3.   ARC_STATE_MRU,
4.   ARC_STATE_MRU_GHOST,
5.   ARC_STATE_MFU,
6.   ARC_STATE_MFU_GHOST,
7.   ARC_STATE_L2C_ONLY,
8.   ARC_STATE_NUMTYPES
9. } arc_state_type_t;
```

代码 2-35

这里带有 ghost 的则对应的是列表的 ghost list 中的数据，这一点在前面已经提到过。而 ANON 则是由一些还没写入磁盘中的数据复制的。在 zfs 中如果是 L2C_ONLY 的话，那么 arc_buf_hdr_t 结构体是会有一些变化的，主要就是 l1arc_buf_hdr_t 这个字段将不再需要了，这一点可以具体查看 arc_impl.h 文件。

最后关于缓存压缩的实现，在 zfs 中是允许压缩和未压缩的数据同时存在的，这里的结构示意图如图 2-18 所示。

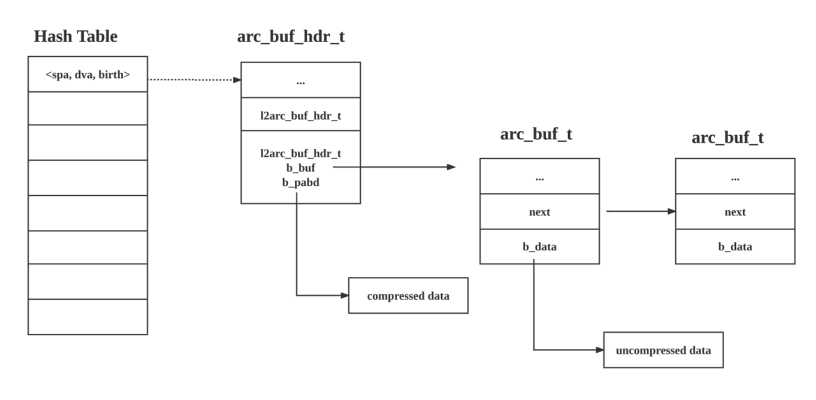


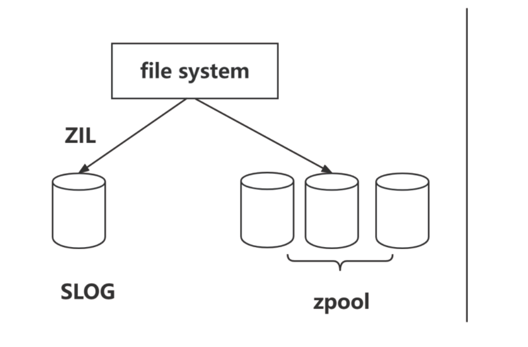
图 2-18

当然关于 zfs 的 arc 实现还有很多细节的，如果感兴趣的话，可以查看源码 arc.c 文件，其中会提到与缓存的锁模型相关的内容，这些也是值得慢慢研究的。

最后可以留意 glusterfs 在使用 openzfs 的时候，推荐的一些设置配置，主要就是开启了 compress 和 arc 参数设置，具体的情况可以参考文档。[8]

2.10 zfs zil

在 zfs 中，除了读取数据是有缓存的，写入数据也是有缓存的，这里就是和 zfs Intent Log（ZIL）有关了。这里需要留意一下 ZIL 和 SLOG 的区别，在 zfs 中，ZIL 是一种机制，而 SLOG 则是一种设备，也就是说 SLOG 是可选的，并不是必需的，但是 ZIL 则是默认的。在存储池中如果没有单独设置 SLOG 设备时，ZIL 机制也是存在的，会在存储池中划分一部分空间出来进行处理。



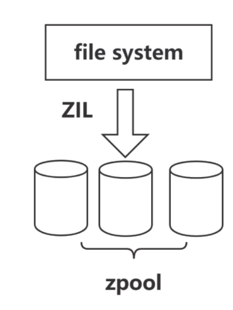


图 2-19

图 2-19 中的左侧部分，就是设置了 SLOG 设备的情况，而右侧则是默认没有设置。那么 ZIL 是做些什么的呢？ ZIL 会在 spa_sync 调用之前，对要操作的数据的日志信息进行落盘，来保

证后续的事务执行出错后能够重放。spa_sync 则是进行事务的数据落盘操作。

对于写入请求中的数据，前面提到过的并不是马上就写入存储设备的，即不会马上持久化，而是会对数据在内存中的缓存进行修改，只有等待一定时间，如几秒或者累积的数据量达到一定程度以后，后台就会自动刷新对缓存中的数据进行落盘，这个就是异步写的过程（write back）。而数据落盘的过程，在 zfs 中是通过事务组（transaction group）来实现的。因此这里引出一个容易被弄混淆的内容，平时我们常见的一些系统调用，如 write 并不会保证数据一定是落盘的，write 返回成功，往往数据只是写入缓存中就立刻返回了。

有些应用系统为了保证重要数据的安全，并不会采用后台异步数据提交落盘的方式，而是会选择同步等待的方式来保证数据写入信息落盘。如调用 fsync 这个命令，则是一种同步等待数据操作信息刷新持久化的请求，这样做会导致性能非常差。因此为了解决这种问题，ZFS 为了提升性能，有一个设备叫作 SLOG（Separate Intent Log），使用固态来加速性能，这些内容都可以在创建 zpool 的时候进行指定。因此使用 SLOG 设备，是为了进一步提升 ZIL 的性能。

这里也需要留意一个细节，fsync 函数的作用是对文件在缓存中的被修改过的数据进行落盘，也就是说，fsync 并不像写请求那样会对缓存数据进行更改，只是发起一个脏数据的刷新落盘动作。

这里简单总结一下，通常来说文件的数据持久化是异步的，但是数据修改的日志记录落盘，一般都是同步等待的，只有这样，才能保证操作记录不丢失。如删除文件时，可以先记录一下删除文件的元信息，把该日志信息持久化到磁盘中，接下来在后台异步删除文件数据，这也是保证事务原子性执行的必要。

ZIL 既然是与日志有关的，那么其到底与 dataset 的关系是怎样的呢？如图 2-20 所示。

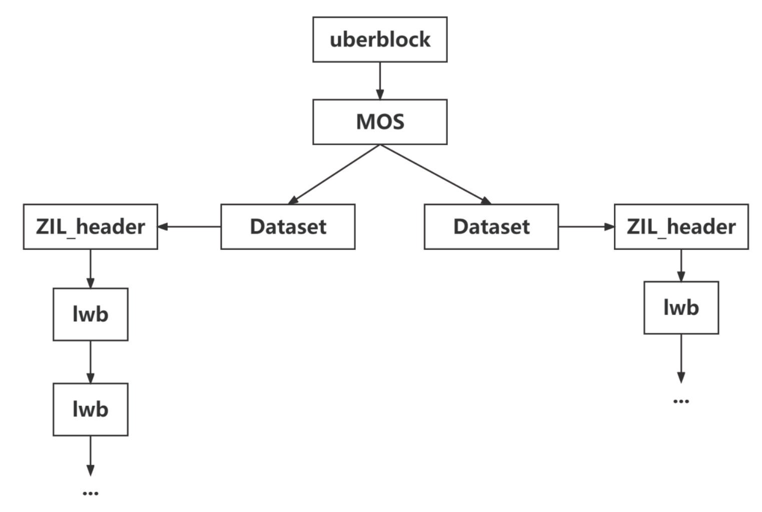


图 2-20

每一个 dataset 都会关联一个单独的 ZIL_header 结构，这个结构是该 Dataset 的日志的头部，而每个 ZIL_header 结构则是用于管理 log 日志项的，而 lwb（log write block）就是每一个日志项记录，用于记录操作的相关信息，在 zfs 中则是用 lwb 结构来进行表示的。

接下来我们来看看 ZIL 的函数调用链。下面从一次正常的 zfs_write 调用请求开始，对于 zfs_write 来说，这里会调用 zfs_log_write，然后会调用 zil_itx_create 和 zil_itx_assign，最后再调用 zil_commit，关系如下所示。

```
1. 1.zfs_write -> zfs_log_write
2. 2.zfs_log_write
3.      ->  zil_itx_create
4.      ->  zil_itx_assign
5. 3.zfs_write → zil_commit
```

代码 2-36

其中 zil_itx_create 会在内存中创建一个 itx 结构，该结构是与 ZIL 的日志落盘相关的，其中 tx 则是事务 transaction 的缩写。zil_itx_assign 则是会把 itx 放在 itxs 的队列中（itxs 是 ZIL 的事务列表，用来记录需要落盘的日志事务）。最后则是调用 zil_commit 进行提交事务处理。

这里 zil_commit 的步骤又有哪些呢？下面来简单描述一下。

（1）找到相同文件对象的异步 itx 一起放到同步事务队列中。

（2）对执行队列中的事务进行落盘。

（3）等待事务落盘完成。

（4）刷新磁盘信息。

（5）通知消息线程。

对于第一点，通常写入数据时都是异步的，而调用了 fsync 这些请求之后，则需要一起提交缓存中的还没刷新同步的脏数据，这时候就可以把同一个文件的一些异步的请求转换成同步请求了，也就是有可能在刷新周期还没到期限时，就进行持久化落盘了。

最后我们再来了解一下 lwb 中的内容到底是什么。lwb 在 zfs 中有一个对应的结构叫作 zilog_t，接着其中有一个结构叫作 zil_header_t，如下所示。

```
1.  struct zilog {
2.   kmutex_t zl_lock;
3.   struct dsl_pool *zl_dmu_pool;
4.   spa_t  *zl_spa;
5.   const zil_header_t *zl_header;
6.   lwb_t  *zl_last_lwb_opened;
7.          ...
8.  }
```

代码 2-37

其中 zl_header 就是每个日志的开头，同时日志结构也是一个链表形式，在 header 后面会关联很多具体的日志信息，例如

创建、写入、truncate 和 attr、链接文件等不同类型操作都有对应的日志类型，如 lr_attr_t、lr_create_t、lr_link_t、lr_write_t 等，对于这些不同操作类型的日志结构字段差异，感兴趣的可自行阅读 zil.h 文件具体查看。

2.11 zfs 事务组

前面已经介绍了 dnode 结构、空间管理 metaslab，还有缓存的内容，当文件被修改之后，不管是同步写还是异步写，都要进行数据落盘，这个过程就涉及了事务组（transaction group，简称 txg），在介绍 txg 之前，先来简单梳理一下文件数据的读写流，还有其中经过的过程等。

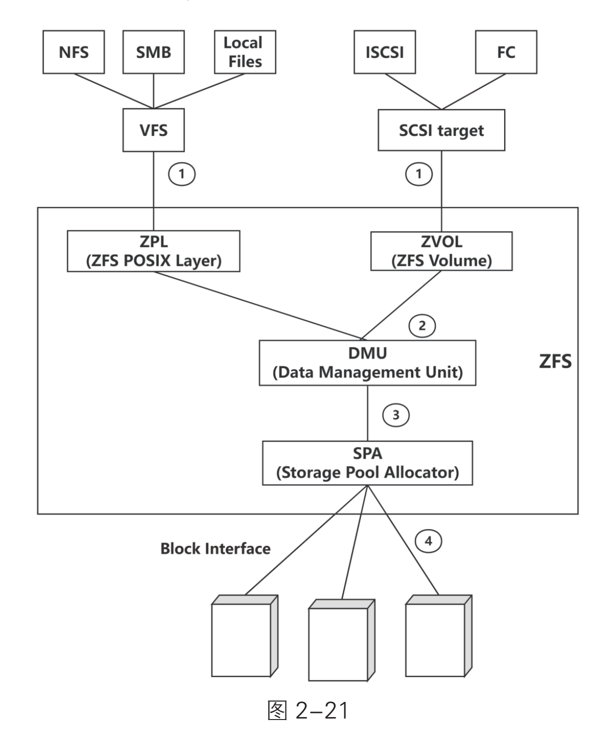


图 2-21

自上而下，不管文件的数据是在 ZPL 还是在 ZVOL 中得到的，二者只是来源不同，但是要实现的语义大部分是相似的，对于文件来说，正常的读写请求和数据经过 ZPL 或 ZVOL 后，会进入 DMU 模块进行处理，这里会对文件的请求进行解析和处理，对于异步写请求，会先写入缓存中，等到了轮转时间周期，再一次性把缓存的数据落盘。而在写入缓存的时候，这里会更新缓存的数据，对于初次打开的文件，被改写的缓存数据会置为 dirty，也就是脏数据，表示缓存中的数据与磁盘中的数据是不一致的。这就是第一和第二步的操作。

这里有一个细节值得关注一下，就是当数据被改写的时候，zfs 因为是 copy on write 的机制，并不会直接改写缓存中的原始数据，而是会写入一个新的地址空间中，然后对父节点层层往上更新信息，包括重新计算校验值 merkle_checcksum，还有修改文件的元信息，包括大小等，更新到文件的 header 重新指向新的地址。该过程如图 2–22 所示。

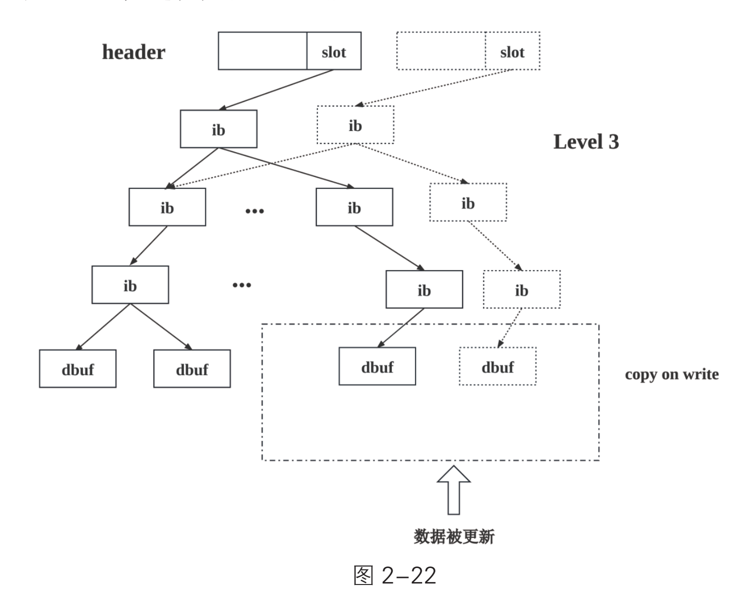


图 2–22

而在数据落盘的过程中，需要先计算好每个文件的脏数据情况，也就是需要统计，当数据落盘时，文件有多少新增和删除的空间地址，还有需要对哪些数据进行计算校验等。这里新增和删除的空间地址，就涉及了 sap 的 metaslab 模块了，还有前面提到的 space map 相关内容，需要申请的空间地址也要进行落盘记录，这就是第三步要考虑的内容。

同时在数据落盘过程中，每个文件的处理都可以看成一个事务，当很多文件都要处理时，批量落盘的文件数据都会在当前的事务周期内进行处理，而这个事务都会绑定在一个事务组中，也就是记录好当前周期内有多少事务需要进行处理，这个就是 txg 相关的内容。而每个 txg 都有一个对应的 id，就是 txg id，该数值类似数据库的表主键那样，是一个递增的数值。

数据的落盘，最终都是要提交给对应的磁盘介质进行处理的，不管是固态硬盘还是机械硬盘，都有对应的驱动，而文件系统则会把该请求发给内核的 IO 模块进行处理，这就是图 2–22 中的第四步的内容了。

在 zfs 中，事务组是 ZFS 对要写入磁盘的数据块进行批处理的方式。zfs 中的 txg 有三种状态，分别是打开（open），静默（quiecsing）和同步（syncing），每个状态有且仅对应一个 txg，下面分别来讲讲这三种状态的不同。

2.11.1 Open

这个状态是当一个新的 txg 创建之后进入的状态，在这个状态中 tx 会绑定到 txg 中，将新事务（对内存中结构的更新）分配给当前打开的 txg。这里始终有一个 txg 处于打开的状态，以便 ZFS 可以接受新的更改（虽然 txg 可能会拒绝新的变化，如果它已经达到了一些极限）。ZFS 将打开的 txg 提前到下一个状态，原因有很多，例如它达到了时间或大小阈值，或者执行了必须

在同步状态下完成的管理操作。

2.11.2 Quiecsing

在 txg 退出 open 状态之后，会进入静默（Quiescing）状态。静默状态的作用是在接受了新的 tx 之后，但是还没真正进行同步的时候提供一定的时间和缓存等待一些 tx 的操作准备。早期设计的时候，是没有考虑过这个状态的，就是只有 open 和 syncing 状态，但是后来增加了。

2.11.3 Syncing

在同步状态中的 txg，这时候就无法接受新的 tx 请求了。那么这里只有当 tx 完成之后才会退出该状态。

同时在该阶段，会处理上一个 syncing 阶段的日志，从前面的提到的内容可以了解，zfs 的空间管理是延迟释放的，因此在当前的 syncing 对应的 txg 中，则会释放前面的 syncing 阶段的日志数据。

下面来分享一下具体在事务组中的事务处理流程，下面以一个写请求为例子。

（1）开启一个 DMU 事务。

（2）修改文件对象，即在缓存中对文件进行增删改查处理。

（3）创建一个 ITX 请求（intent log transaction），即创建对应的事务日志请求来描述事务内容，其中包括文件的偏移量（offset），修改内容、修改时间和对应的事务 id（txg）等信息，具体见 zil_itx_create 函数。

（4）将 ITX 分配给对象集的 ZIL，具体见 zil_itx_assign。

（5）调用 dmu_tx_commit 提交完成文件事务，并且通过回调 zil_commit 函数来返回用户请求。

同时还要留意，对于数据的写入有异步写和同步写的区别，其中同步写是需要阻塞等待事务返回的，而异步写则是先写入缓存中，等待数据累积到一定阈值或者时间周期轮训的时候进行持久化落盘。这里不管是异步写还是同步写，都是要创建对应的 DMU 事务的。

对于 ZIL 中的日志，如果没有单独设置 SLOG，也就是没有单独设置日志盘，则会默认在数据盘中划分一段空间来当作日志数据空间，这样处理的原因是因为日志数据都是比较小的，同时受限于 DMU 中的脏数据设置的比例和周期轮转时间，当数据需要落盘持久化的时候，所产生的日志数据并不会很大，通常数十吉字节大小即可。因为日志数据很小，若频繁分配或释放，会造成严重的空间碎片化，如果与数据盘的数据空间混合着使用，则会非常影响性能且导致磁盘使用达到一定比例后，出现无法分配大的连续空间问题，因此设置单独日志磁盘空间则可以相对规避该问题。同时还要留意，为了避免日志数据出现问题，也需要对日志的数据进行校验，但是该校验码是保存在自身的日志数据中的，也即自校验的方式（self–checksuming）。

这里也同时产生了一个新的问题，日志的提交是批量的吗？为了解答该问题，可看图 2–23。

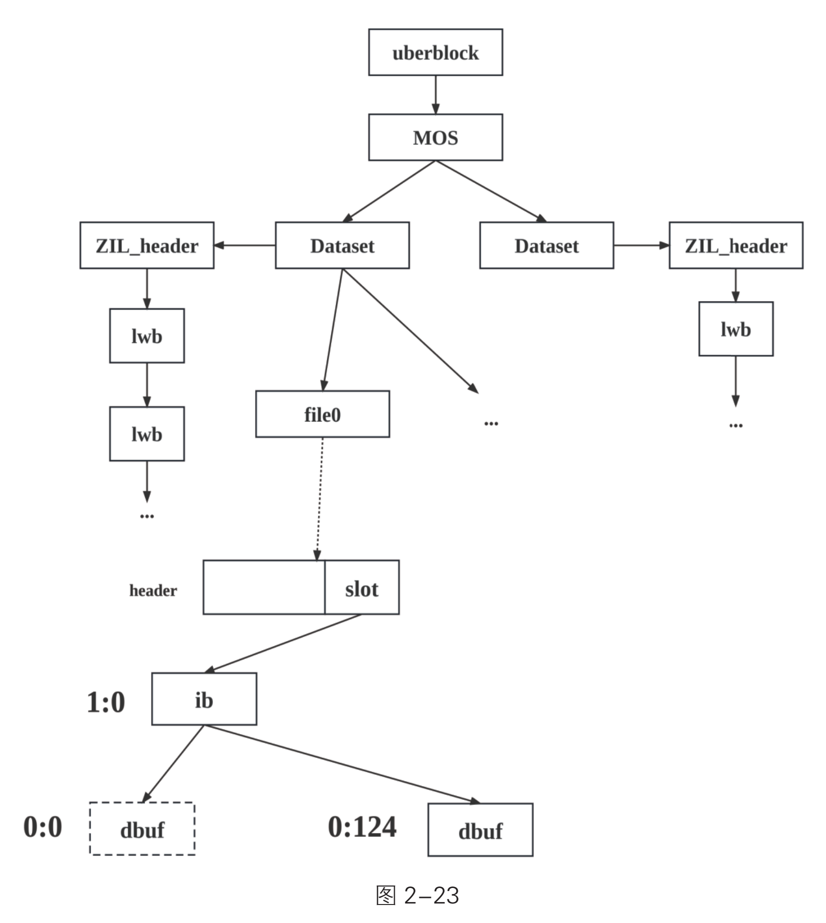


图 2-23

图 2-23 是结合了文件的结构，dataset 对应的 zil 模块的示意图。当事务组要更新持久化数据时，若自下而上的更新，会更新多个 dataset 中的文件数据，还有提交每个文件对应的 zil 日志数据，最后还要更新 uber 空间。若每个文件都是单独提交的，即完成了文件事务的操作后，单独提交日志并且更新 uber 空间，则会出现并发冲突且效率很慢。因为从前面的内容可以了解，每个 ZIL 中都有一个 header，每个 header 下面会有当前事务组的多个事务日志数据。单独提交则会把一个顺序写变成了随机

写入，对于机械盘来说，这样的效率和延迟是非常不理想的。为了解决这些问题，就会考虑等待多个文件都完成了数据提交之后，再进行批量的日志提交，这样就是一个比较高效的顺序写。因此对于 zil 的持久化，当每个文件的事务完成后，提交日志数据给 zil 后就会进入休眠等待，然后等 zil 进行批量提交日志数据再唤醒处理，这个就称为 ZIL-LWB 的超时机制。

当然这是以前的做法，在最新的 zfs 版本中引入了持久化内存（pmem），因为 pmem 的性能很高，zfs 官方通过测试发现，大量的时间等待耗费在了日志的批量提交上面，因此也对上述过程进行了优化，感兴趣的人可以自行阅读论文 *Low-Latency Synchronous IO For OpenZFS Using Persistent Memory*。

2.12 读请求流程代码走读

前面一直提到与写请求有关的内容，如 dirty data 和 ZIL，而在写入之前，需要先读取数据，因此下面来了解一下在 zfs 中读取数据的流程。本小节的内容是以 openzfs 中的 2.1.x 代码为例，如果后续有变更，建议各位读者可以根据 commit log 来回溯调用过程，又或者以未来的最新代码来自行调整理解。

本小节内容会涉及源码走读，对于源码的阅读，会相对比较枯燥且无趣，但是对于理解了基本概念以后，深入理解 zfs 是非常有必要的，同时源码里面会包含非常多的处理细节，而细节则可以真正考验一个系统的水平。很多时候，我们遇到的很多开源项目，在读很多架构概念和内容时，看起来都是差不多或者直观上看不出明显区别，而真正的差异，往往隐藏在处理细节上面，尤其是对一些极端情况的考虑，还有技术边界要考虑清晰，这是一个优秀的系统所必需的。

首先来了解一下读取数据的入口，也就是 zfs_read 函数。

```
1. int
2.     zfs_read(struct znode *zp, zfs_uio_t *uio, int ioflag, cred_t *cr) {
3.       ...
4.         zfs_locked_range_t *lr = zfs_rangelock_enter(&zp->z_
   rangelock,
5.           zfs_uio_offset(uio), zfs_uio_resid(uio), RL_READER);
6.       ...
7.       zfs_rangelock_exit(lr);
8. }
```

代码 2-38

这里 zfs_read 函数中的参数 znode 封装了要读取的文件信息，而 zfs_uio_t 则是读取的信息的回填地址，长度等，ioflag 是 O_SYNC 标志。

对于文件的读写，这里需要有一个概念，那就是区间锁 rangelock，所谓区间锁，就是要读写文件的范围。对于一个正常的文件读写，这里有三种情况，分别是读、写和追加。

而读写通常是不会改变文件结构大小的，但是追加是有可能的。因此这三种情况是互斥的，也就是说，可以把追加也理解为一种特殊的写入请求。而读写请求是需要上锁的，这里读取的流程是使用 RL_READER，写入是 RL_WRITER，追加则是 RL_APPEND，可以查看 zfs_rangelock_type_t 中的字段。

这里为什么需要加入区间锁呢？因为读写是可以并发的，只要不是对相同区间的数据进行读写。同时这里有一个地方需要留意，要读写的区域是否会有冲突，取决于要读取的 offset 和长度是否有重叠的 block，也就是说，虽然可能一个读请求和写请求的 offset 不相同，但是因为数据是有重叠的 block，这也会产生冲突。如下所示，下面是两个读取请求，中间会有重叠区域，这里就被称为 overlap，对于读取是可以有重叠的，这里就与读写锁的理解是类似的。

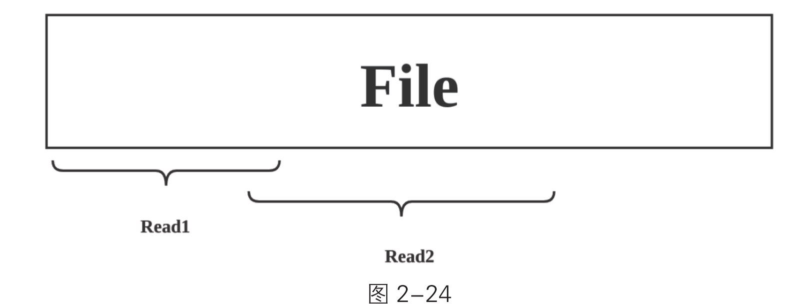


图 2-24

在拿到了区间锁之后，就要计算读取什么内容了。根据前面的内容可以了解，文件通常是树形结构的，而读取的时候，除了读取数据，还需要考虑读取其父节点，因为在读取了数据之后，其放入了缓存里面，后续可能会对读取的数据进行修改，成为脏数据，脏数据的内容，则需要重新计算该节点及其父节点的相关信息，以此来层层递增向上刷新数据。

```
1. int
2. dmu_buf_hold_array_by_dnode(dnode_t *dn, uint64_
   t offset, uint64_t length,
3.     boolean_t read, const void *tag, int *numbufsp, dmu_buf_
   t ***dbpp,
4.     uint32_t flags)
5. {
6.           ...
7.           if (read)
8.             zio = zio_root(dn->dn_objset->os_spa, NULL, NULL,
9.               ZIO_FLAG_CANFAIL);
10.
11.           ...
12.       for (i = 0; i < nblks; i++) {
13.             dmu_buf_impl_t *db = dbuf_hold(dn, blkid + i, tag);
14.             ...
15.
```

```
16.          if (read) {
17.              (void) dbuf_read(db, zio, dbuf_flags);
18.              if (db->db_state != DB_CACHED)
19.                missed = B_TRUE;
20.              }
21.             dbp[i] = &db->db;
22.        }
23.         ....
24.
25.         if (read) {
26.           //等待异步读取完成
27.           err = zio_wait(zio);
28.                  ...
29.          }
30.
31. }
```

代码 2-39

该函数中的调用则是从 zfs_read 函数中的 dmu_read_uio_dbuf 调用的。该函数的处理中，前面还有一些标记的处理，也就是 dbuf_flags 的内容，主要就是考虑是否有加密的内容，如果想要了解，可以自行查看 DMU_READ_NO_DECRYPT 等字段内容。

接着这里还有一个值得注意的内容就是 zio_root 函数，该函数封装了读取请求的内容，这里可能会读取多个不同的 block，读取的时候需要等到这些读取完成才可进行，也就是代码中的 zio_wait 内容。

对于 zio_root，这和 zio tree 有关，因为 zfs 是有存储池的设计的，因此读写请求可能会涉及多个磁盘，而把这些请求封装成一棵树的形式，头部添加一个 null 的标记，以此来证明结束标记。

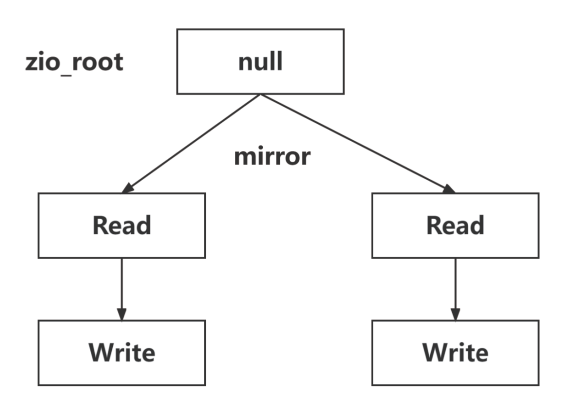


图 2-25

dbuf_hold 中的内容就是前面提到的要计算获取其父节点的内容，这里返回的是一个 dmu_buf_impl_t 结构体，该结构体的内容会传递给 dbuf_read 函数进行最终的读取。而在 dbuf_hold 函数中会调用 dbuf_hod_impl 函数，该函数里面则调用 dbuf_find 和 dbuf_findbp 函数，对应上述流程，如下所示。

```
1.  int
2.  dbuf_hold_impl(dnode_t *dn, uint8_t level, uint64_t blkid,
3.      boolean_t fail_sparse, boolean_t fail_uncached,
4.      const void *tag, dmu_buf_impl_t **dbp)
5.  {
6.      ...
7.      db = dbuf_find(dn->dn_objset, dn->dn_object, level, blkid);
8.      if (db == NULL) {
9.         ...
10.        err = dbuf_findbp(dn, level, blkid, fail_sparse, &parent, &bp);
11.        ...
12.         db = dbuf_create(dn, level, blkid, parent, bp);
13.     }
14. }
```

代码 2-40

了解了要读取哪些内容后，接下来就要真正考虑读取数据了，这里有一个问题需要考虑，数据到底在哪里呢？因为读取

请求需要读取的数据，可能在缓存中，也可能不在，也就是说可能在读取数据之前已经被缓存过了，例如在 L1 和 L2 的 ARC 中，这时候读取就只要从缓存中读取即可。同时还有一些情况，一部分的数据已经在缓存中，一部分并没有。因此这里关于缓存的状态管理，就涉及 dmu_buf_impl_t 中的字段 db_state，如下所示。

```
1. typedef enum dbuf_states {
2.  DB_SEARCH = -1,
3.  DB_UNCACHED,
4.  DB_FILL,
5.  DB_NOFILL,
6.  DB_READ,
7.  DB_CACHED,
8.  DB_EVICTING
9. } dbuf_states_t;
```

代码 2-41

该枚举中的不同数值就代表了缓存的不同状态，这里状态轮转关系如下所示。

```
1.      +----> READ ----+
2.      |    |
3.      |   v
4.       (alloc)-->UNCACHED       CACHED-->EVICTING-->(free)
5.      | ^ ^
6.      |   |   |
7.      +----> FILL ----+   |
8.      |       |
9.      |       |
10.     +--------> NOFILL -------+
```

代码 2-42

其中 DB_SEARCH 根据 openzfs 中的内容说明可以了解到，这是和 dbuf 释放有关的。

在了解了 db_state 的缓存状态轮转关系之后，如果要读取的数据，一部分在缓存，一部分没有在缓存时，那么 db_state 的状态是不能设置为 CACHED 的，需要等待读取的全部缓存命中之后才能进行设置。同时这里需要留意，如果有多个线程来请求相同的区间进行读取数据时，最终只能有一个线程来执行读取任务，其他线程需要等待返回即可。

对于 dbuf_read 函数，这里最终调用的读取数据是在 dbuf_read_impl 函数中，调用 arc_read 函数进行读取，如下所示。

```
1.  static int
2.  dbuf_read_impl(dmu_buf_impl_t *db, zio_t *zio, uint32_
    t flags,
3.      db_lock_type_t dblt, const void *tag)
4.  {
5.     ...
6.     (void) arc_read(zio, db->db_objset->os_spa, &bp,
7.       dbuf_read_done, db, ZIO_PRIORITY_SYNC_READ, zio_flags,
8.       &aflags, &zb);
9.     ...
10. }
```

代码 2-43

在调用 arc_read 函数的入参中，会带上 dbuf_read_done，如果读取成功了，这是一个状态设置的函数，会最终把缓存设置为 DB_CACHED，同时这还有一个标记是 ZIO_PRIORITY_SYNC_READ，这个是 ZIO 的 IO 优先级标记，是 zio_priority_t 枚举中的数值。根据 openzfs 的官方文档内容 [6] 可以得知，同步读的优先级是最高的。

```
1.  int
2.  arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp,
3.       arc_read_done_func_t *done, void *private, zio_priority_
    t priority,
4.       int zio_flags, arc_flags_t *arc_flags, const zbookmark_
    phys_t *zb)
5.  {
6.      ...
7.      if (*arc_flags & ARC_FLAG_WAIT) {
8.        cv_wait(&hdr->b_l1hdr.b_cv, hash_lock);
9.        mutex_exit(hash_lock);
10.       goto top;
11.     }
12. }
```

代码 2-44

上面代码中的 b_cv 则是 l1arc_buf_hdr 结构体中的字段，用于进行前面提到重复读取线程的回调，这个调用根据代码中的注释内容可以了解到，是在 arc_read_done 中，感兴趣的自行阅读即可。

```
1.  int
2.  arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp,
3.       arc_read_done_func_t *done, void *private, zio_priority_
    t priority,
4.       int zio_flags, arc_flags_t *arc_flags, const zbookmark_
    phys_t *zb)
5.  {
6.      ...
7.      rzio = zio_read(pio, spa, bp, hdr_abd, size,
8.        arc_read_done, hdr, priority, zio_flags, zb);
9.    acb->acb_zio_head = rzio;
10.
11.   if (*arc_flags & ARC_FLAG_WAIT) {
12.      rc = zio_wait(rzio);
13.    goto out;
14.   }
15. }
```

代码 2-45

到这里为止，与 DMU 相关的内容已经了解得差不多了，而请求提交到 zio 之后，则是 ZIO 模块中相关的内容，为了避免代码走读逻辑过长，因此想再深入了解 ZIO 模块的，则可以看下一小节的内容。

而 DMU 中，其实还有一些其他的处理细节需要留意一下，例如当读取数据的请求正在执行的时候，来了一个写入相同位置的请求，那又该如何处理呢？这里其实就要区分不同情况了，如果读请求并没有提交到 ZIO 中，那么在 dmu_read 函数中则可以处理这种请求，取消掉读取请求，直接把写入请求的数据放到缓存中。而如果请求已经提交到 ZIO 中，则可能无法拦截成功，因为 ZIO 最终的任务提交，是要交给内核进行处理的，那时候已经不是文件系统的内容了。

另外这里还有一个细节值得考虑，那就是关于文件层级过大，也就是 overflow 的理解。如果要读取的文件 level 层级过大（例如目前默认最大是 5），那么该文件的结构可能就需要调整，主要就是调整了间接块的大小，让其可以指向更多的子块，以此来减少文件层级，减少数据读取遍历的次数，具体的情况可以在 dmuf_findbp 函数中了解到。

最后需要再次明确一下，如果有写入请求时，遇到的是非对齐写，也就是写入的数据大小，不是完整的一个 block，例如数据块是 4k 大小，但是只写入 2k 时，这时候数据需要先从磁盘中加载再写入。但是遇到了对齐写，则不需要从磁盘中加载该数据块，只要在缓存中分配空间直接写入新数据即可。所以，在很多系统软件中会比较倾向于对齐写，这可以减少 IO 读取次数，同时也可减少一些异常情况，例如非对齐写中，如果不对数据块做初始化处理，那么可能会出现一些计算校验码不一致的情况，因为数据块的剩余部分可能会有其他数据的存在。因此也有一些系统软件在写入数据时，如果数据块没有对齐写，

则会进行补 0 操作，填充数据达到对齐写的效果，如图 2-26 所示。

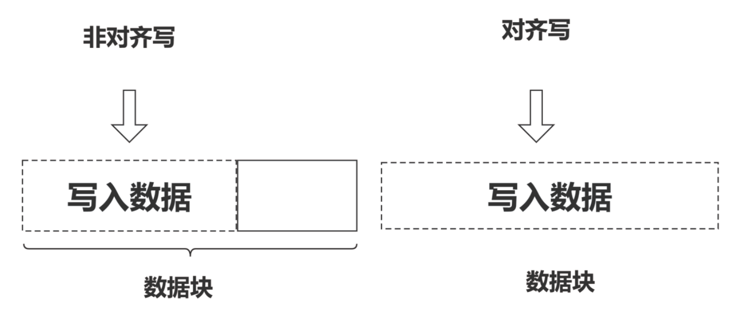


图 2-26

2.13 zfs zio

本小节的内容是接着上面的读请求介绍的，是当 DMU 模块的 zio 提交到 zio 模块之后开始的。

如果读取的数据不在缓存中时，则需要从磁盘中读取数据了，这里则是与 zio_read 和 zio_wait 有关了。其中 zio_read 则是封装了 zio_t 对象，给 zio_wait 进行调用。而 zio_wait 函数中则会调用 zio execute 相关的函数，对 zio pipeline 进行设置 stage 状态，这与 zio_pipe_stage_t 结构有关，如下所示。

```
1.  static zio_pipe_stage_t *zio_pipeline[] = {
2.    NULL,
3.    zio_read_bp_init,
4.    zio_write_bp_init,
5.    zio_free_bp_init,
6.    zio_issue_async,
7.    zio_write_compress,
8.    zio_encrypt,
9.    zio_checksum_generate,
10.   zio_nop_write,
11.   zio_ddt_read_start,
12.   zio_ddt_read_done,
```



```
13.   zio_ddt_write,
14.   zio_ddt_free,
15.   zio_gang_assemble,
16.   zio_gang_issue,
17.   zio_dva_throttle,
18.   zio_dva_allocate,
19.   zio_dva_free,
20.   zio_dva_claim,
21.   zio_ready,
22.   zio_vdev_io_start,
23.   zio_vdev_io_done,
24.   zio_vdev_io_assess,
25.   zio_checksum_verify,
26.   zio_done
27. };
```

代码 2-46

这里涉及了很多不同的内容，其中 ddt 是和重复数据删除有关的，而读取数据流程中有关的则是 zio_read_bp_init，zio_vdev_io_start，zio_vdev_io_done 等内容。其中 zio_vdev_io_start 还会考虑数据是从哪个磁盘中读取的，因为 zfs 中是有存储池设计的，如果是镜像，那么相同的数据则会存在多个磁盘中，如果从其中一个磁盘中读取失败了，则需要从其他的磁盘中读取。如果是 RAIDZ 的存储池处理则也是类似的。感兴趣的可以自行阅读一下如 vdev_mirror_io_start 和 vdev_raidz_io_start 函数。

当然不管是 mirror 还是 raidz 的存储池，在其 io start 函数中，都会调用到 zio_nowait 函数，其中入参的有 zio_vdev_child_io 函数。

```
1.  static zio_t *
2.  zio_vdev_io_start(zio_t *zio)
3.  {
4.      ...
5.      if (vd->vdev_ops->vdev_op_leaf &&
6.        vd->vdev_ops != &vdev_draid_spare_ops &&
7.        (zio->io_type == ZIO_TYPE_READ ||
8.        zio->io_type == ZIO_TYPE_WRITE ||
9.        zio->io_type == ZIO_TYPE_TRIM)) {
10.               ...
11.              if (zio->io_type == ZIO_TYPE_READ && vdev_cache_read(zio))
12.               return (zio);
13.              if ((zio = vdev_queue_io(zio)) == NULL)
14.               return (NULL);
15.      }
16.
17.      vd->vdev_ops->vdev_op_io_start(zio);
18.      return (NULL);
19. }
```

代码 2-47

回到 zio_vdev_io_start 函数中，会把 zio 放在 IO 队列中，也就是调用 vdev_queue_io，调用对应的 io start 函数，也就是 vdev_op_io_start。

在 vdev_op_io_start 函数中会调用 zio_interrupt，则是当磁盘加载了数据之后，会中断 cpu 对其进行回调的，这里 zio pipeline stage 则是调用了 zio_vdev_io_done 函数。

同时在该函数中，可以看到 zio_type_t 结构体的其他类型，如下所示。

```
1.  static void
2.  vdev_disk_io_start(zio_t *zio)
3.  {
4.        ...
5.        switch (zio->io_type) {
6.      case ZIO_TYPE_IOCTL:
7.                  switch (zio->io_cmd) {
8.              case DKIOCFLUSHWRITECACHE:
```



```
9.                  error = vdev_disk_io_flush(vd->vd_bdev, zio);
10.                 }
11.          ....
12.      }
13.
14.      error = __vdev_disk_physio(vd->vd_bdev, zio,
15.      zio->io_size, zio->io_offset, rw, 0);
16. }
```

代码 2-48

这里有一个分支是调用 vdev_disk_io_flush 的，这个是和写入请求的脏数据有关的。当有写入请求的数据到缓存之后，系统是会定时或者达到脏数据的一定比例之后就调用一次 flush 请求，把脏数据持久化到磁盘中。

接着则是与 bio 相关的内容了，具体地可以查看 vdev_disk_physio 函数的内容。对于读取的数据，加载之后还需要进行计算校验，这时候就是 zio_pipe_stage_t 中的 zio_checksum_verify 函数功能了。

最后来查看一下和 ZIO 队列有关的参数，这里的参数定义在 vde_queue.c 文件中，其中 zfs_vdev_max_active，zfs_vdev_sync_read_min_active 等参数是值得关注的，具体的参数数值和含义，可以见该文件的内容，这里就不再重复了。

2.14 zfs 磁盘移除

在 zfs 中，有存储池的存在，一般新建的时候会添加很多磁盘，有添加也就会有移除，对应的就是 zpool remove 命令。当然，在磁盘移除命令中，要和磁盘掉线区分开，前者是正常情况下的磁盘移除该 zpool，而后者则是某个设备被热插拔了，但是配置信息应该是还要留在系统中的。同时对于磁盘替换命令，本质上是一个组合命令，就是磁盘添加和磁盘移除的组合，一般

是先添加一个新的磁盘，格式化设置好之后，再进行移除操作。而添加磁盘相对简单，但因为是一个空的磁盘，则移除磁盘比较麻烦，里面会存在各种数据，同时数据可能还在读写等，因此要考虑的场景和问题相对复杂很多。

首先要考虑什么条件下的磁盘可以被移除？

如果有异常的磁盘，正在修复数据的时候，能否会被移除呢？还有一些严重警告时能会被否移除呢？这时做这些操作，都可能会导致一些未知的异常风险，严重的时候甚至会出现数据丢失的可能。通常在操作磁盘移除时，建议在 zpool 没有异常情况下进行。

另外对于移除的磁盘，是什么类型的磁盘都可以被移除吗？Raidz 和 Mirror 的可以吗？在 openzfs 中，对于 RaidZ 的磁盘是不允许被移除的，这里是有限制的。原因也很简单，因为 RaidZ 本质上是对数据进行分片后再进行校验，通常移除了一个磁盘以后，可能会导致磁盘数量不满足条件等，因此对于 RaidZ 类型的存储池磁盘，目前是不支持移除的，当然后续不确定是否会有新的变化，关于这一点，建议各位要密切关注自己所使用的版本的最新要求。

关于 RaidZ 类型的磁盘无法被移除这一点，不只是 zfs 有这样类似的要求，对于 glusterfs，EC 冗余卷，也是不允许在线升级的，[8] 这点要求的限制也是类似的。

除了以上内容，还要考虑一点，被移除磁盘的数据需要重新找地方写入，如果是多副本的存储池，那么会把数据写入其他磁盘下，这时候需要保证剩余的磁盘空间容量足够存放这些数据，否则会导致无法写入，如图 2–27 所示。

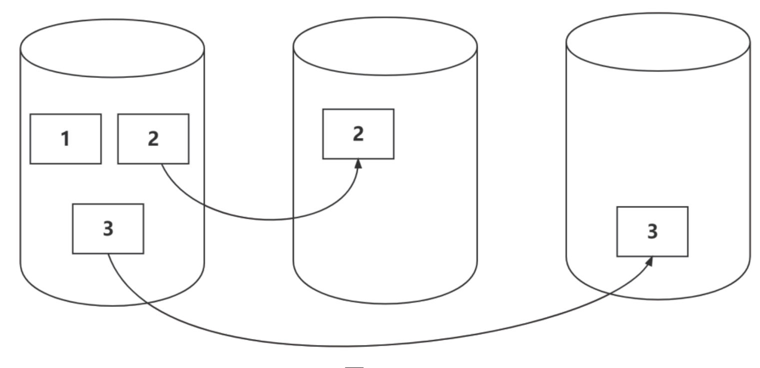


图 2–27

移除磁盘过程中，需要考虑和进行哪些步骤呢？其代码逻辑过程主要在 vdev_removal.c 文件中，这里主要会涉及两点：禁止新的请求和数据迁移。下面开始简单分享一下。

- **禁止对被移除磁盘的写入**

这一点比较好理解，因为要移除磁盘的时候，不再允许对新的读写请求进行分配，这一点主要是在 spa_vdev_remove_top 函数中进行的。当然在执行该函数之前，有一次检查的功能，也就是 spa_vdev_remove_top_check 函数所需要进行的功能了。需要注意的是，这新的空间申请之前，需要确保 zpool 中不会只有一个磁盘，否则是无法进行数据迁移的。

- **数据迁移**

磁盘卸载的时候，要进行数据迁移，在 zfs 中是需要启动一个新的线程来进行处理的，这和 spa_vdev_remove_thread 函数有关，对于这个过程，需要知道要迁移的数据在哪里。这时候 MetaSlab 和 space map 就可以显示其作用了。zfs 中的 metaslab 就是记录了空间分配的情况，而 space map 则是日志记录。而数据迁移的过程，可以理解为数据同步，也就是 sync task，这是一

种数据内部重新分配写入过程，因此可以看成是一次特殊的写入请求。因此从 MetaSlab 中获取到了哪些数据要重新迁移，接下来就是进行数据拷贝复制了，产生新的迁移记录，这里采用 space map 来记录。

对于加载被移除磁盘的空间分配记录，可以见如下代码所示。

```
1.  static __attribute__((noreturn)) void
2.  spa_vdev_remove_thread(void *arg)
3.  {
4.        for (msi = start_offset >> vd->vdev_ms_shift;
5.         msi < vd->vdev_ms_count && !svr->svr_thread_exit; msi++) {
6.                  if (msp->ms_sm != NULL) {
7.      VERIFY0(space_map_load(msp->ms_sm,
8.          svr->svr_allocd_segs, SM_ALLOC));
9.                     }
10.       }
11. }
```

代码 2-49

上面的代码省略了很多的内容，在该函数中，会使用 spa_vdev_removal 结构中的 svr_thread 字段来记录，这里调用 space_map_load 来加载读取要被移除磁盘的 MetaSlab 信息。

接着其延伸出一个小细节，每次读取多少数据呢？一般来说，磁盘数据的迁移过程，应该是越快越好，因为迁移时间太长，会影响到正常的线上生产环境的性能，甚至有其他突发的高可用问题，例如磁盘坏盘或者掉线等，会影响性能甚至造成数据损坏。因此为了加速数据迁移，在 zfs 中，对于读取的数据长度，有一个变量可以进行限制，也就是 SPA_MAXBLOCKSHIFT，这个数值目前在 zfs 中是 16M，也就是说，每次读取的数据量以 chunk 为单位，最多读取 16M 为一个 chunk。

对于这样的优化，有其明显的优缺点，优点很明显，就是数据分片越大，一次性读取会比小分片的相对速度快得多，当

然也不能太大，例如以 GB 为单位的话，明显就过大了，同时也超出了一般磁盘的 IO 性能。

同时这样做的缺点也很明显，当一次性读取 16M 的数据时，如果数据地址空间分配是连续的，那么是比较适合的，但是在一些特殊情况下，可能会导致性能不太理想，也就是文件的空间分配碎片化比较严重时，如图 2-28 所示。

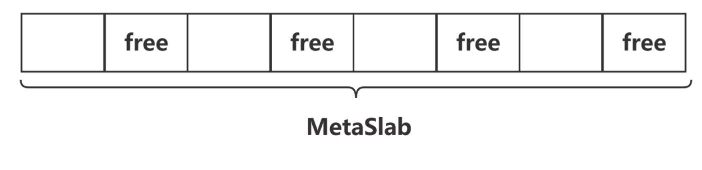


图 2-28

对于要复制的 chunk，如果其中有比较多的删除记录，那么这对于数据迁移是不太友好的，会明显降低数据迁移的性能。

同时，这里还要考虑一个问题，如果申请一个连续的大分片空间，在要写入的磁盘中并没有那么大的连续空间该如何解决呢？这时候就需要考虑对分片数据进行切分了，也就是分成两份甚至多份数据，如图 2-29 所示。

、

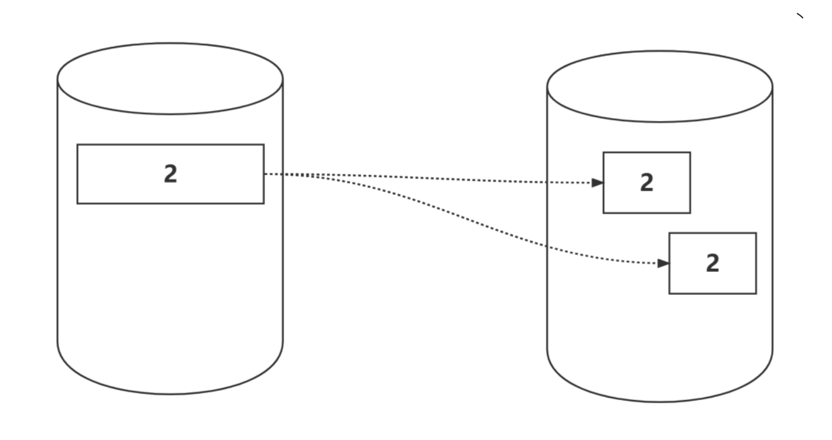


图 2-29

同时还要考虑，在数据迁移过程中，如果还有其他的请求进入被移除磁盘中，这些该如何处理？请留意一下，这里并不是指新的请求，而是指那些已经被接受的请求，包括读写和删除这些。对于读取请求还比较好处理，因为磁盘还没有正式被移除，因此也是可以读取到数据的，而写入请求会被重定向到新分配的空间。但是对于删除请求，这里需要考虑两种情况。

（1）要删除的数据并没有被复制到新的磁盘空间，这时候直接在被删除磁盘中操作就可以了。

（2）如果一旦要删除的数据之前已经被复制到新的地址空间了，那么这时候需要再次发起一次新的删除请求到新分配的地址，这就是 free_from_removing_vdev 函数需要考虑的。

最后对于磁盘移除的过程，zfs 也是提供了取消的命令，这个可以从 zpool 命令中找到相关的参数，这里就不再讨论了。而取消移除的过程，则需要反向操作一次，也就是对移除新分配的数据，进行删除操作，并且重置磁盘信息状态等。

2.15 zfs scrub

在 zfs 中有两个比较有趣的命令和参数设置，就是 dedup 和 scrub，其中 dedup 是 deduplication 的简称，也就是重复数据删除技术，这个技术的出现是为了节省空间，而这个技术的出现也比较早，在 2009 年的时候 Jeff Bonwick 先生的博客就已经讨论过这个技术的实现。[10] 而 scrub 则是进行数据完整性校验的命令，这个命令不只是在 zfs 文件系统中有，在一些固态硬盘中也可能会有对应的命令。

首先我们来讨论一下，为什么需要考虑一个这样的命令呢？一般来说，文件系统写入的数据到磁盘设备，不管是机械盘还是固态，通常应该信任其数据是可靠的，但是有时候往往会有一些额外的情况，例如数据很久没有读写，属于冷数据，那么在固态里面，可能会出现比特反转的现象，这时候就需要使用 scrub 命令来进行检查并且修复数据了。下面来简单感受一下该命令的使用。

```
1.  ~# zpool  scrub hilton
2.  ~# zpool  status
3.    pool: hilton
4.   state: ONLINE
5.    scan: scrub repaired 0B in 00:00:00 with 0 errors on Sat Feb  4 17:41:02 2023
6.  config:
7.          NAME        STATE     READ WRITE CKSUM
8.          hilton      ONLINE       0     0     0
9.            vdc       ONLINE       0     0     0
10.           vdd       ONLINE       0     0     0
11. errors: No known data errors
```

代码 2-50

这里需要留意的是对 scrub 命令进行操作时，zfs 中对其的优先级是最低的，也就是比同步和异步读写的优先级都要低，这一点可以查看 zfs IO scheduler 的相关内容，如表 2-4 所示。

表格 2-4

优先级	IO 类似	相关参数
高 | | \/ 低	同步读 (sync read)	zfs_vdev_sync_read_min_active zfs_vdev_sync_read_max_active
	同步写 (sync write)	zfs_vdev_sync_write_min_active zfs_vdev_sync_write_max_active
	异步读 (async read)	zfs_vdev_async_read_min_active zfs_vdev_async_read_max_active
	异步写 (async write)	zfs_vdev_async_write_min_active zfs_vdev_async_write_max_active
	scrub	zfs_vdev_scrub_min_active zfs_vdev_scrub_max_active

对于 zfs 的 IO 类型，这里需要和我们常见的 Linux IO 概念区分开，我们常说的 Linux IO 是指对文件的操作 IO 请求，但是其中可能会包括了增删改查等操作，或者是一些扩展属性命令设置等，这些都是 Linux IO 请求。而 zfs 的 IO，主要讲的就是文件的同步或者异步读写请求，zfs 在缓存中对文件进行修改的操作，会对文件的修改进行持久化落盘。其中提交的文件修改，是封装成 linux 的 bio 请求，最后提交给对应的磁盘设备进行操作。

对于 IO 请求的性能研究问题，通常有多个不同的层次，首先是文件系统在缓存中对文件的修改 IO 操作，还有是在文件系统修改完成之后，提交到内核 IO 调度模块，写入磁盘设备的 IO 操作等。所以读者在遇到 IO 性能测试或者相关问题时，需要根据语义语境来判断具体指代哪种情况。

目前 zfs 官方正在考虑增加一个叫"zhack scrub"的命令，主要是希望在用户态中使用 scrub，关于该命令的 pr，[10] 可以自行查看最新的进展。

2.16 zfs dedup

第一个问题是为什么需要考虑重复数据删除技术。从这个全称中就可以知道，很多时候数据是重复的，例如发送邮件的时候，附件经常都是重复的。如果可以应用 dedup 技术，就可以大大地减少空间了。与此同时，又产生了一个新的问题，到底在文件系统，dedup 该在哪个层面去实现呢?

因为对于文件系统来说，文件结构是树形结构的，这里实现该技术就有三个层面的路线了，分别是文件级别的、block 级别的还有字节级别的。这三个不同级别的实现都会带来不同的考虑。

如在 block 级别实现的话，这里的灵活度就更高了，只需要了解匹配两个文件的 block 块是否一致就可以了，而 zfs 最终选择的也是在 block 块级别来进行的。而字节级别的话，这里因为需要一个个字节进行对比，这样的工作量是非常庞大且不适合的。

决定了在什么级别实现 dedup 技术之后，接着产生了另外一个问题，什么时候进行 dedup 技术？也就是说，到底什么时候对文件进行判断是否有重复。在文件写入的时候还是写入完成之后。写入完成之后进行处理，这里可以平衡一下机器的负载，主要是当机器空闲的时候进行操作。异步重复数据消除通常用于 CPU 能力有限或多线程能力有限的存储系统，以对日间性能的影响降至最低。考虑到足够的计算能力，同步重复数据消除更可取，因为它不会浪费空间，也不会对现有数据进行不必要的磁盘写入。zfs 考虑到未来的发展，在选择了数据写入时，同步进行 dedup 技术。因此在 openzfs 的源码中，其文件 zstream_redup.c 就是处理这些的。

而在 zfs 中，如果想要开启或者关闭 dedup 技术，只需要执行如下命令即可。

```
1. zfs set dedup=on tank
2. zfs set dedup=on tank/src
```

代码 2-51

其中 tank 是假设已经存在的 zfs pool。当然，如果只需要对其中某个目录进行设置。

其还会产生一个问题，到底如何判断两个文件的 block 是真的一致呢？这里通常判断文件是否被修改过，可以使用 checksum 算法，也就是 hash 算法来考虑，而更加严谨地思考一下，hash 算法到底能否保证不会产生冲突呢？这就取决于 hash 算

法的实现或者是否相信 hash 算法的结果了。而通常使用的 hash 算法有 SHA256。因此当用户不再信任，或者想要更加严谨一点的话，这时候就需要一种机制来进行保证，一旦 hash 算法真的出现冲突了，也可以解决这个问题。这里的解决方案就是进行字节对比。在 zfs 中有一个选择就是解决这个问题的。

```
1. zfs set dedup=verify tank
2. zfs set dedup=fletcher4,verify tank
```

代码 2-52

把 dedup 设置为 verify 之后，哪怕两个 block 的 checksum 是一致的，也会进行字节对比，以此来保证出现冲突后可以发现。当然在 zfs 中也提供了一些其他的 hash 算法来进行 dedup 技术的判断。但是根据官方的建议，通常不建议修改对应的 hash 算法。

还需要考虑一个问题，是否需要额外的内存空间来记录重复的数据信息。因为在实现 dedup 技术时，需要有一张表来记录重复数据，这张表在 zfs 中被称为 DDT，而在日常使用中，开启了 dedup 之后，对性能和内存要求是很高的，根据 oracle 的建议，这里需要先判断一下，内存是否满足开启该功能，计算的规则如下。

```
1. In-core DDT size (1.02M) x 320 = 326.4 MB of memory is required.
```

代码 2-53

也就是说，如果 DDT 表的大小是 1.02M，那么这里内存应该要是这张表的 320 倍以上，因此这就是为什么不会经常开启 dedup 的原因了。当然，对于 dedup 功能来说，其实也是可以使用快照和克隆来代替的。同时，一旦开启了 dedup 之后，在删除数据和数据迁移的时候，也需要有更多的考虑，其增加了代

码复杂度和场景复杂度，性能也会随之可能下降。

2.17 zfs 快照

文件系统的快照功能是一个比较常见的内容，同时也算是大部分文件系统都应该具备的基础功能之一，但是快照的实现方式，则是有不同的考虑的。总的来说，目前笔者遇到过依赖 Linux 内核的快照，具体来说就是依赖 lvm2 来实现的，其以 glusterfs 为代表。简单来说就是快照功能的实现，并不是通过自身来实现，而是依赖于操作系统。当然也有一些是自己负责快照的，zfs 就是其中一个。

实现快照，需要和虚拟机的快照进行区别，常用的 vmware 和 virtualbox 中也有快照功能，但是虚拟机中的快照是需要暂停运行的，也就是 stop the world，而 zfs 中实现的快照，并不需要暂停服务。

实现快照，要解决的第一个问题是以什么内容为基准，判断数据是否应该保留下来呢？这个问题很好解决，那就是文件中的 txg id，当然在 zfs 中是使用 birth time 来称呼的（见 blkptr_t 结构）。为什么使用这个名称呢？因为在文件的读写过程中，需要不断记录当前操作文件的最新的事务 id，因此以该 id 来进行衡量，其是不断变化的，所以需要有一个固定的事务 id 来进行记录，这个就是 birth time，可以成为文件生成时事务 id。在数据库中实现的事务级别隔离，也是基于事务 id 来实现的，在 mysql 中通常称为高低水位。

zfs 对于数据的更新是写时复制的（copy on write），这里创建了快照之后，需要保留 root block，防止数据被删除，同时会记录一个 snapshot time（简称 snap time），如图 2-30 所示。

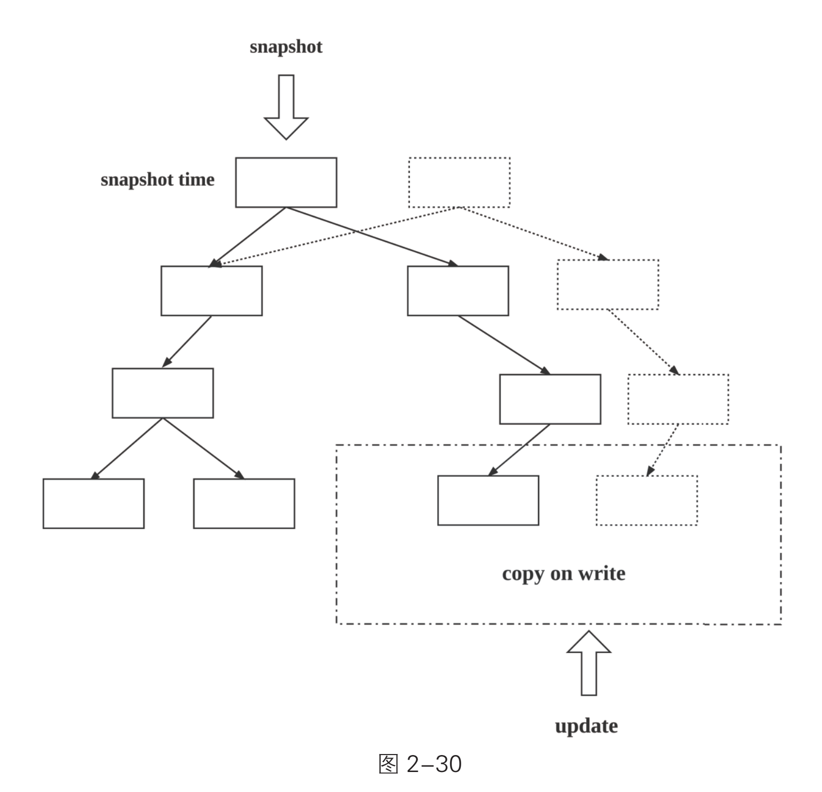


图 2-30

有了 snap time 之后，当后续如果要删除快照时，需要知道该 block 是否被快照引用，如果 block 的 birth time 大于最近一次快照的 snap time，则证明该 block 是在快照执行后生成的，那么则可以删除该 block。

对于文件系统来说，快照是可以创建多个的，因此每一个快照都会有其对应的 snap time，删除快照的时候，需要知道该数据是否被当前快照独占，因为只有独占的数据块才能删除，否则该数据还被之前创建的快照所需要，被是不能删除的。

同时对于快照的使用需要留意，不能过于频繁地创建和删除快照，因为在快照被创建后，每次对文件的处理还需要多一些额外的逻辑校验判断，即需要考虑该文件是否有快照，如果

需要保存的话，还不能进行删除，其会影响到数据落盘的性能等。而 zfs clone 功能原理也是和快照非常相似的，区别就是克隆之后的数据可以进行读写修改，但是快照目前默认是只读的。同时克隆和快照还有一点不同，克隆还需要额外记录修改过的快照文件，因此还需要增加一个 time 来记录对应的修改情况，关于这点感兴趣的可以查看相关资料。

2.18 zfs 动态 trim

zfs 中还有一个比较有趣的高级特性，就是动态 trim 功能的实现（autotrim），这个功能的实现主要是为了优化 zfs trim 的问题，其为手动执行 trim 的时候，命令执行时间较长，会阻塞住其他的用户请求，导致系统可能会出现被阻塞的现象。当然，为了了解这个功能，我们需要先从 trim 到底是什么讲起。

在过去几年中，固态硬盘越来越受欢迎。从中我们可能读过或至少听过其他人谈论 SSD（固态硬盘）与传统硬盘相比的速度有多快。如果已经在使用 SSD 或想要购买 SSD 以提高计算机性能，通常 trim 的支持是必不可少的。trim 是一个命令，通过该命令，操作系统可以告诉固态 SSD 哪些数据块不再需要，可以删除，或者标记为可自由重写。换句话说，trim 是一个命令，它帮助操作系统准确地知道要移动或删除的数据存储在何处。此外，每当用户或操作系统发出删除命令时，trim 命令立即擦除存储文件的页面或块。这意味着下次操作系统尝试在该区域中写入新数据时，不必先等待删除。简单来说，就是 trim 功能的出现优化了固态在数据擦写方面的性能。

在 zfs 中也是支持 trim 功能的，下面先简单感受一下命令参数的使用，如下所示。

```
~# zpool trim
missing pool name argument
usage:
        trim [-dw] [-r <rate>] [-c | -s] <pool> [<device> ...]
```

代码 2-54

当然，zfs 目前最新的版本也已经开始支持了动态 trim 的功能，可以进行如下设置查看是否支持。

```
~# zpool set autotrim=on
missing pool name
usage:
        set <property=value> <pool>

the following properties are supported:

        PROPERTY             EDIT   VALUES

        autotrim              YES    on | off
        ...
The feature@ properties must be appended with a feature name.
See zpool-features(7).
```

代码 2-55

从上面的命令参数可以看到这里支持开关，同时在 zfs 中对该命令的支持是同时支持 autotrim 和手动执行，二者并不冲突。

这里有了 zfs trim 之后，为什么还要考虑 autotrim 呢？其中一个重要的原因是 trim 功能是阻塞的，因此手动执行的 trim 命令所需要的时间更长，有时候是无法接受的，因此就产生一个想法，能否在后台自动运行 trim 功能，这就是 autotrim 来源之一。

在后台运行 trim 功能（autotrim），首先要做的就是启动一个程序来定时运行 trim 运行功能，同时这里还要考虑一个问题是不能影响到正常的服务运行，因此这需要控制住运行的速度，也就是不能一次性对多个磁盘进行操作，而 zfs 对于 autotrim 的优化，考虑的颗粒度更加细腻。从前面的内容可以知道，zfs 对

于每个磁盘的管理是使用 metaslab 进行划分区间的，因此使用 autotrim 的时候，为了更好地控制速度，默认是每次只对一个磁盘的一个 metaslab 进行执行，这并不会阻塞太多其他的 metaslab 的使用。

当然，对于 trim 的优化，如果启动了 trim 之后，能否进行取消或者挂起呢？这当然是可以的，这也就是 zpool trim 命令 --rate 和 --cancel 这些参数的作用了。同时为了更进一步地观察 trim 的执行情况，zfs 中如果开启了该动态 trim 功能之后，可以通过命令来观察，命令如下所示。

```
~# zpool iostat -r   hilton

hilton      sync_read     sync_write     async_read     async_write    scrub    trim
req_size     ind    agg    ind    agg    ind    agg    ind    agg    ind    agg    ind    agg
----------  -----  -----  -----  -----  -----  -----  -----  -----  -----  -----  -----  -----
512      0    0    0    0    0    0    116   0   4     0    0    0
1K     0    0    64    0    0    0    183    6    4    0    0    0
2K     0    0    0    0    0    0    33    60    0    2    0    0
4K     0    0    0    0    0    0    6    32    1    0    0    0
8K     12    0    20    0    0    0    0    2    0    7    0    0
16K     0    0    0    0    0    0    0    0    0    2    0    0
32K     0    0    0    0    0    0    0    0    0    0    0    0
64K     8    0    32    0    0    0    0    0    0    0    0    0
128K     0    0    8    0    0    0    0    0    0    0    0    0
256K     0    0    0    0    0    0    0    0    0    0    0    0
512K     0    0    0    0    0    0    0    0    0    0    0    0
1M     0    0    0    0    0    0    0    0    0    0    0    0
2M     0    0    0    0    0    0    0    0    0    0    0    0
4M     0    0    0    0    0    0    0    0    0    0    0    0
8M     0    0    0    0    0    0    0    0    0    0    0    0
16M     0    0    0    0    0    0    0    0    0    0    0    0
--------------------------------------------------------------
```

代码 2-56

代码在实现 trim 功能的时候，zfs 需要对 spa 增加一个和当前进行 trim 相关的内容地记录了，这个就是在 metaslab 中的 ms_trim，如下所示。

```
1.  struct metaslab {
2.      ...
3.      range_tree_t *ms_freeing;
4.      range_tree_t *ms_defer[TXG_DEFER_SIZE];
5.      range_tree_t *ms_trim;
6.      uint64_t ms_disabled;
7.  }
```

代码 2-57

在 metaslab 中有 ms_disabled 参数，该参数的作用就是希望阻塞住一些用户修改请求时的使用。trim 在实现时，需要记录哪些信息呢？这里就是对应 vdev_trim.c 文件中的 trim_args 结构体，如下所示。

```
1.  typedef struct trim_args {
2.
3.    vdev_t  *trim_vdev;
4.    metaslab_t *trim_msp;
5.    range_tree_t *trim_tree;
6.    trim_type_t trim_type;
7.    uint64_t trim_extent_bytes_max;
8.    uint64_t trim_extent_bytes_min;
9.    enum trim_flag trim_flags;
10.
11.   hrtime_t trim_start_time;
12.   uint64_t trim_bytes_done;
13. } trim_args_t;
```

代码 2-58

其中需要知道哪个磁盘的哪些空间需要进行 trim，也就是对应 trim_vdev，trim_msp 和 trim_tree 这三个参数，trim_type 记录的命令是发起的还是动态 trim 的类型，同时还有控制的速度，

也就是和 trim_extent_bytes_max，trim_extent_bytes_min 有关的内容，最后还有启动和完成的时间信息等。

如果启动了动态 trim，在 spa 中会有一个标记 spa_autorim_t 的枚举为 SPA_AUTOTRIM_ON，同时启动的后台线程函数为 vdev_autotrim_thread，而进行 trim 的时间，则是在 txg 中的事务执行完成之后，在 metaslab_sync_done 函数中进行判断，如下所示。

```
1.  void
2.  metaslab_sync_done(metaslab_t *msp, uint64_t txg)
3.  {
4.    ...
5.    if (spa_get_autotrim(spa) == SPA_AUTOTRIM_ON) {
6.    range_tree_walk(*defer_tree, range_tree_add, msp->ms_trim);
7.    if (!defer_allowed) {
8.     range_tree_walk(msp->ms_freed, range_tree_add,
9.         msp->ms_trim);
10.   }
11.  } else {
12.   range_tree_vacate(msp->ms_trim, NULL, NULL);
13.  }
14. }
```

代码 2-59

关于 trim 功能更具体的细节，建议感兴趣的可以自行阅读 zfs 的源码，主要内容在 vdev_trim.c 中，如 trim 功能停止的函数为 vdev_autotrim_should_stop。

最后提醒一下，因为 trim 的使用不可避免地会影响到正常 IO 性能，在生产环境下使用时，建议先对日常数据流量做好对照测试，运行一段时间后观察性能的具体影响情况。



3.文件存储从单机到分布式

前面提到了单机文件系统部分，其主要以 zfs 为主，单机文件系统在发展了多年后，分布式文件系统的兴起和对象存储的出现，对于单机文件系统来说，是否是一个挑战，还是一个替代呢？因此我们把目光从单机转移到分布式系统上面来思考一下到底分布式领域需要考虑哪些问题？又有哪些差异呢？下面来一起了解一下。

3.1 单机到分布式的架构变化

我们需要思考第一个问题，为什么需要分布式？这个问题相信对于拥有多年工作经验的人来说，都会有非常多的答案，大体来说就是分布式可以保证单机物理机在出现断电或者意外的物理损坏后，数据在分布式冗余下可以得到安全保障。同时分布式系统的出现，还可以对系统进行横向扩展，因为单机系统不管性能多么优越，都会受到单机物理硬件的限制，如磁盘空间，cpu 核数和内存空间等，因此分布式的出现可以极大地扩展存储的发展。

3.1.1 元数据中心架构

当然对于分布式文件系统来说，分布式文件系统，是不是单纯地把多个单机节点组成一个集群来进行管理呢？这里当然不是那么简单，其如图 3–1 所示。

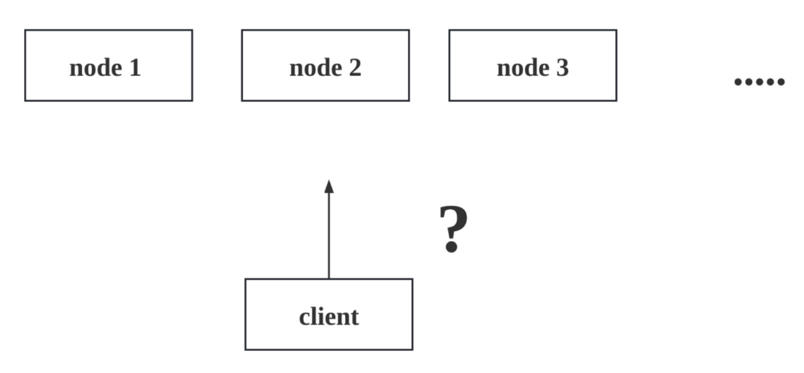


图 3–1 客户端访问

这里需要解决一个问题，就是客户端如何知道要访问集群哪个节点，也就是数据到底保存在哪里呢？为了解决这个问题，目前有两种架构，一是找一个地方保存文件的相关路径信息等，其中包括一些元数据信息，如文件的大小、权限和名称等，当客户端第一次需要访问的时候，先知道文件所在的节点信息，其就可以直接访问该节点，保存的地方，通常就被称为元数据中心，或者元数据节点。

从图 3–2 可以看到，集群中的多个节点都会向元数据中心连接上报，文件进行创建的时候，在节点 node N 中创建成功之后，则上报文件的元数据信息到元数据节点中（假设称为 MetaServer），同时每个节点的存活状态都要上报，通常情况下数据是多副本冗余的，文件通常使用三副本保证数据安全，当其中一个节点掉线时，则还可以访问其他剩余的副本节点进行读写数据，这就是大家所熟悉的多数一致性原则（quorum）了。

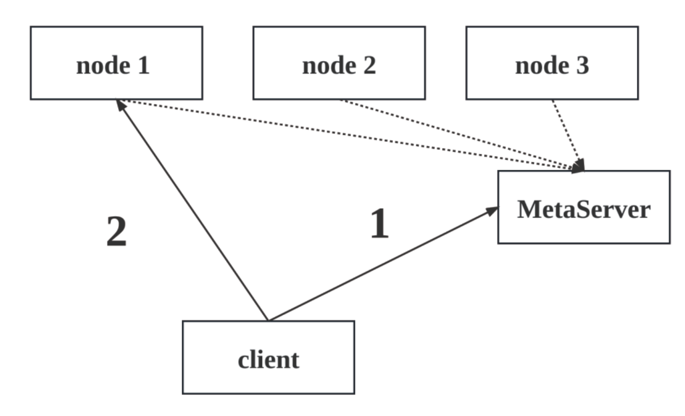


图 3-2 元数据节点

随着数据的不断增多，可能三五个节点也不足以保存下数据了，需要不断增加节点，同时元数据节点可能也会是一个瓶颈了，因此这里就需要产生数据分组的概念了，如图 3-3 所示。

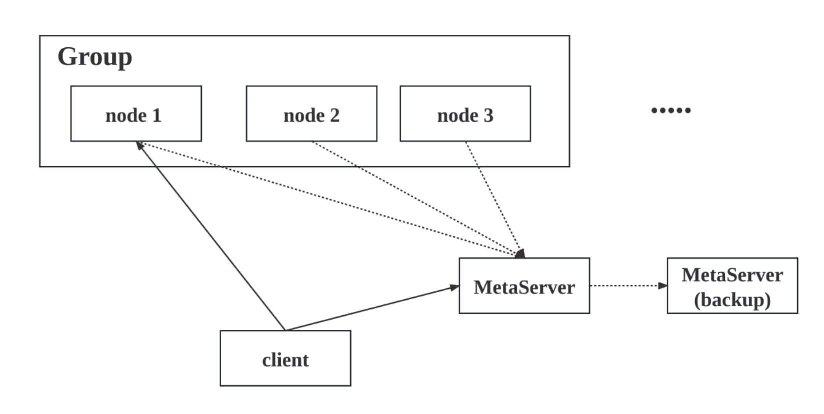


图 3-3 数据分组

为了保证元数据节点的数据安全，这时候也需要有一个备份节点来进行同步，在元数据宕机出问题之后能够保证集群的平稳切换，这里以 HDFS 为代表，其中 NameNode 和 SecondaryNameNode 的概念出现了。

当然数据的同步与备份，往往又会受到效率和时间的影响，有可能正在同步备份的数据还没完成，但是元数据节点已经宕机了，那么这时候也可能会出现元数据丢失的现象，但这可以通过重试等机制来进行解决。

随着集群的发展，元数据中心担任的任务很重，包括探测节点存活和保管文件元数据信息等，因此为了减轻元数据节点的负担，一些系统会把监控任务剥离出来。当出现网络隔离的时候，若元数据节点与其他节点的网络异常了，但是其他节点之间的网络是正常的，即元数据节点被网络隔离时，会导致节点的存活性判断存在问题，这时候引入监控中心节点则可以解决该问题，如图 3-4 所示。

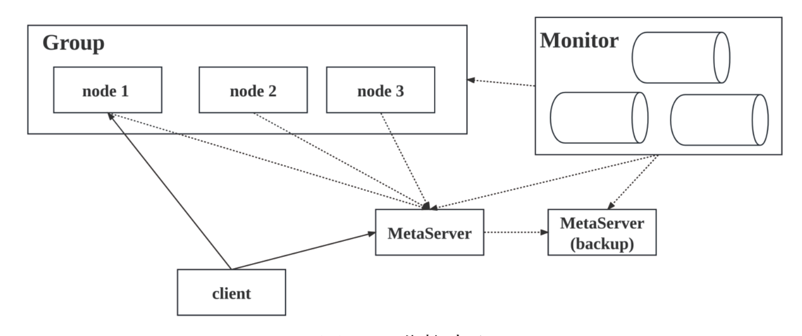


图 3-4 监控中心

对于监控集群中心的节点，通常以奇数为主，主要也是为了方便达到一致性判断。即当一个节点判断另外一个节点异常或者请求不可达时，需要通过监控中心的节点连接判断，若监控中心的多数节点都无法连接到该异常节点，则可以上报确认该节点已经异常了，而 ceph 则是这种架构类型的代表之一。

3.1.2 无中心架构

前面提到分布式系统的一些架构发展，是为了解决数据存放的访问问题，因此引入了元数据节点，也进一步延伸出了很

多其他的模块，包括监控中心等。当然目前分布式系统中还有另外一条发展路线，就是无中心架构的出现，一个文件节点是否存在，并不一定需要元数据中心，其中以 glusterfs 为代表。

在 glusterfs 中，每个数据的分组是以 volume 为基准的，每个 volume 就是一个数据组，可以是副本，也可以是 EC 卷，每个 volume 之间的元数据在集群中每个节点都会被保存的，因此实现了元数据中心的部分功能。当然从该架构的设计可以了解，该架构的设计，随着数据的增长，之后若需要进行扩容，则会相对麻烦，若 volume 信息较多时，需要集群每个节点都更新保存，因此数据的读取效率相对没有上述架构高效。

glusterfs 的无中心架构，也有其优势，相对适合大文件数据的保管，因为其管理和配置得相对简单，没有引入过多的节点类型，适合一些冷数据和大文件数据的保存。同时也适合一些相对数据规模较小的公司团队进行运维。

另外为了解决 glusterfs 集群的部分缺陷，有公司团队会在 glusterfs 集群之上，使用 etcd 或者自研系统来增加一层元数据层，以达到快速访问集群元数据的效果，如图 3–5 所示。

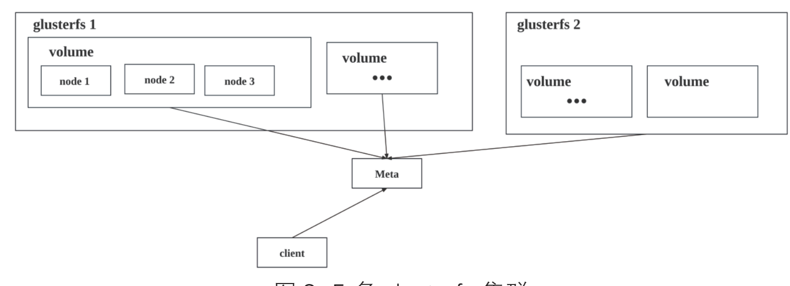


图 3–5 多 glusterfs 集群

以图 3–5 为例，其有多个 glusterfs 集群，对于客户端来说，并不知道底层的数据到底分布在哪个集群中，同时每个集群也可能会有多个不同的 volume，通过一些集群层面的数据分配策略，把数据存放在多个不同集群中，方便一些冷热数据迁移和

数据重平衡等。

最后提示一下，对于 glusterfs 的使用，笔者更加建议用于大文件数据和冷数据，并不建议在日常的生产系统上使用 glusterfs 作为后端存储。同时自 2023 年起，k8s 方面目前已经不再支持 glusterfs 地使用了，关于这点，建议大家留意。

3.2 分布式下的一致性

对于分布式系统，我们除了要关心其架构，还要考虑分布式系统的一致性，对于常见的副本模式来说，通常三副本是遇到比较多的一种情况，有了多副本之后，读写时的一致性问题也就随之出现了。但是在考虑一致性问题之前，我们需要先来看看客户端的数据传递方式问题，也是为了便于深入理解数据一致性问题。

3.2.1 数据广播和流式传递

首先，我们需要思考一个问题，客户端向节点写入数据时，是向往三个副本的数据节点写入，还是写入主节点（leader），然后由主节点往后传递到其他节点（follower）呢？

下面先来看看这两种方式的区别和优缺点。

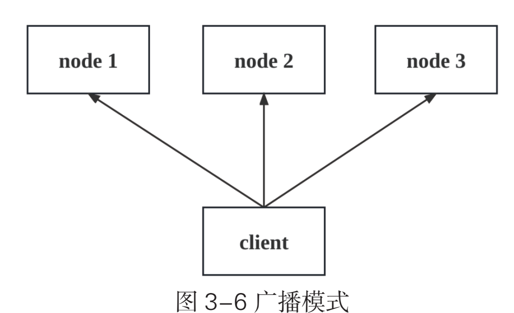


图 3–6 广播模式

图 3–6 所示的方式被称为广播模式，数据的广播模式就意

味着客户端需要自行向多副本节点发送数据，并且做好消息传递的可靠性与稳定性的管理，例如当其中一个副本节点出现掉线宕机的情况，或者消息异常要进行重试等，相对来说，这样的方式比较直接明了，但是优缺点也会比较明显。

其中比较明显的弊端之一就是客户端发送写入请求时数据传递数量很大，导致占用客户端的流量出口带宽。因为存储和数据库的数据类型不同，数据库是一条条 SQL 语句，例如 select 等，请求中并没有带上大量的数据，但是存储中需要携带大量的写入数据，例如视频文件、图片、压缩文件等，这些数据可能较大，有些甚至会达到几十上百吉字节大小，而若都由客户端向数据节点发送数据，那么需要客户端把发送三次数据到节点中（如未特别指明，多副本通常默认三副本），其对于客户端的出口带宽流量占用则比较大。同时还要注意到客户端和数据节点之间的网络并不一定是在同一局域网内，因此可能还会受到其他网络距离的影响等。

当然广播模式的好处也是非常明显的，关于其好处，了解完流式传递的弊端后再进行分享，这样会有更加深刻的理解。

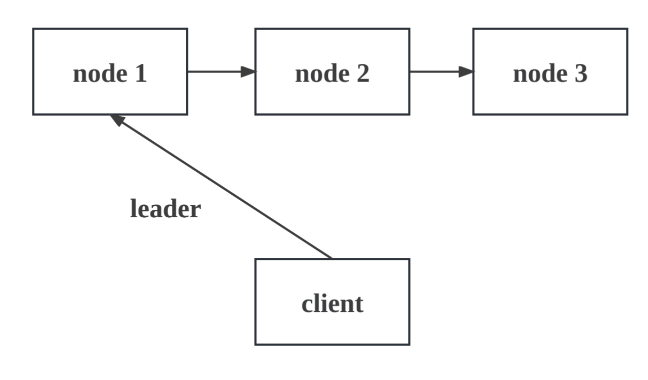


图 3–7 流式传递

图 3–7 所示的模式被称为流式传递，其中客户端需要先向

配置中心了解整个系统的拓扑结构，哪个节点是主节点（leader，也有一些被称为 master/slave，但是这里默认称为 leader），客户端向 leader 节点发送请求和数据，交由 leader 节点向其他两个副本节点发送数据（follower），最后成功后由 leader 节点返回写入成功的消息报文。

流式传递的模式，需要有一个系统配置中心来管理整个系统的拓扑结构，通常会使用到 zookeeper 或者 etcd 这些开源系统，需要额外增加多一个组件或者模块功能来维护，同时也就意味着对于这样的模式，并不适合使用无中心架构系统。

流式传递的好处也是非常明显的，首先在数据写入的时候，由 leader 节点来向后传递，可以解决客户端向数据节点传递数据的出口带宽占用问题，leader 与 follower 节点之间通常都是一个局域网内的服务器节点，并且服务器机房的网络带宽也是相对较高的，因此数据传递的速度往往会比较快。

这种模式的出现，还可以解决多客户端并发写入同一个文件的问题，该问题的出现主要是因为若有多个客户端同时写入一个文件时，在广播模式下，有可能会因为写入的数据到达数据节点的先后顺序不同，导致写入的数据被相互覆盖掉，例如当 client 1 要写入数据文件 file1 的数据为 a，当 a 刚写入 node2 时，可能才达到 node1，与此同时，如果此时有另外一个客户端 client2 也要写入该文件，数据为 b，那么若情况相反，即数据 b 刚写入 node1，准备到达 node2，这时候就会使两个客户端之间的数据相互覆盖了。

对于这个问题的解决方法也很简单，只要连接打开文件时申请到文件修改权限即可。下面先来看看流式传递的客户端并发情况，如图 3–8 所示。

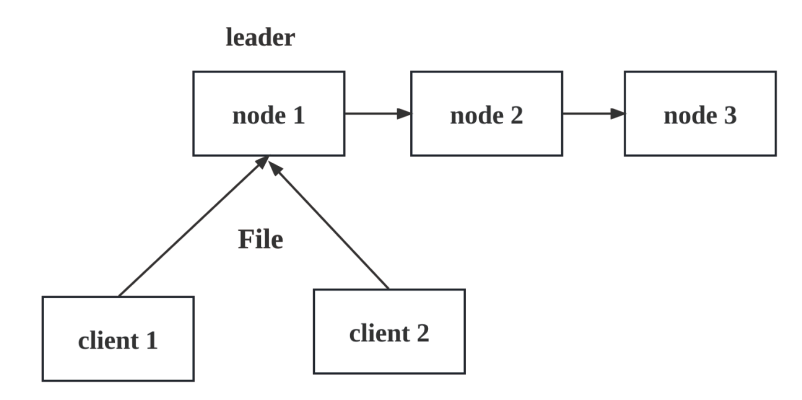


图 3-8 流式传递客户端并发冲突

若出现了两个客户端的并发写入时，leader 节点可以通过获取当前文件的信息，如知道当前是 client 1 已经申请到该文件的写入权限，那么可以暂时拒绝接受 client 2 的文件写入和修改。

若 leader 节点的网络出现故障，或者节点宕机的话，客户端节点是无法感知到正在写入的文件数据在其他节点的情况，需要进行 leader 切换之后，根据配置中心节点选择出新的 leader 节点，重新连接获取相关信息才能重新知道文件的具体情况，这点也是广播模式带来的优势之一了。同时若网络频繁抖动的话，也有可能会出现频繁的 leader 切换，在极端情况下，也会带来复杂的文件异常问题，对于文件的修复也是一个非常麻烦的过程。关于分布式系统下的数据文件异常修复问题，后面再具体分享。

不管是流式传递还是广播模式，其实并不是固定的，也就是说两种模式之间是可以进行切换的，若数据进行流式传递，leader 发现其中一个 follower 节点有问题时，可能就会自动切换到广播模式了。同时对于 leader 往 follower 发送数据，这里也有两种不同的方式，一是 leader 向下游其中的一个 follower 节点发送，该节点再向下游发送；二是 leader 广播向其他所有的 follower 节点发送数据，如图 3-9 所示。

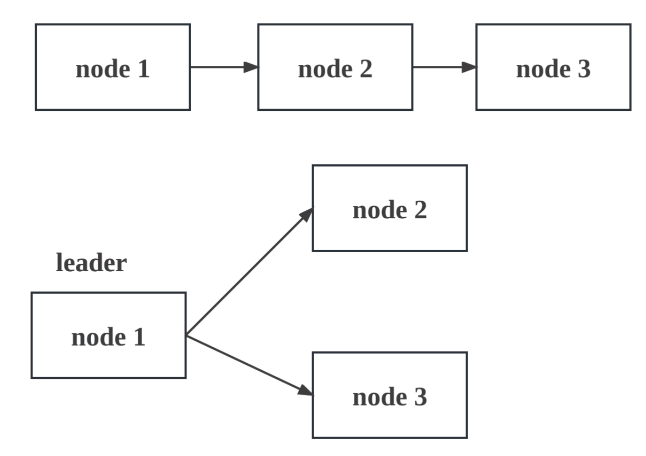


图 3-9 leader 往 follower 发送数据模式对比

对于这两种方式的差异，有了前面提到的内容，相信大家也能很好地理解二者之间的优缺点。同时各位也可以带着这样的内容去感受一下不同的开源系统项目，在这方面的实现和具体细节差异，相信会有更深刻的理解和认识。

3.2.2 多数一致性

在了解了客户端广播和流式传递模式差异之后，接下来关注一下分布式数据的一致性问题，但是在了解该问题之前，需要先来了解一下一次完整的写入请求，在分布式系统中的流程，client 先向 MetaServer 节点获取到系统的拓扑结构图后，主要流程如下。

（1）client 找到 leader 节点，打开文件并且写入数据。

（2）leader 节点接收到请求后，向 follower 节点传递并写入数据，同时也写入本地。

（3）写入成功后，leader 节点向 MetaServer 发送文件元数据信息，成功后回复客户端节点数据写入成功。

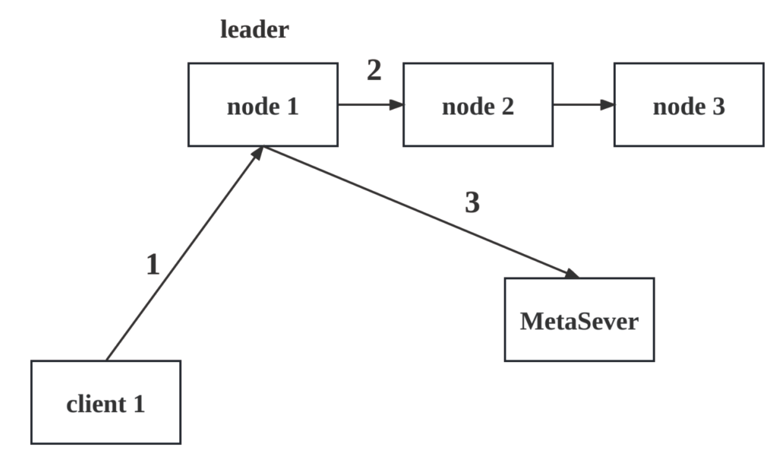


图 3-10 数据写入流程

这里对于第二点，当 leader 节点向 follower 节点中写入（图 3-10 中的 node2 和 node3 时），其就产生了一个问题，leader 的数据向 follower 传递之后，还有本地也要写入，那么这里是否要等待所有的写入都完成了才算成功呢？当然不是，通常只要多数节点成功之后即可返回，也就是大家常听到的 quorum 原则，对于三副本来说，只要写入其中两个副本节点（通常是 leader 本地和其中一个 follower 节点）即可认为写入成功了。

这里有一些小细节需要留意，通常来说，数据写入本地是比较快的，因此 quorum 条件的满足，大部分情况下是本地写入成功了，再加上其中一个 follower 节点写入成功即可，但是也有可能当 leader 节点有大量数据请求，或者本地写入的速度较慢时，会影响其他 follower 节点的完成情况。这种情况的出现，会使 leader 节点的数据落后于 follower 节点，客户端读取文件的时候，leader 节点也并不一定都是最新的数据。

这里还需要留意，向 leader 和 follower 节点写入数据，写入成功是指把数据写入哪里呢？通常情况下，一般的写入请求都是异步写入，也就是数据写入缓存中即可认为成功，若数据是

三副本，每个节点的数据会定时或者周期累积刷新持久化到磁盘，会增加数据的冗余来保证安全性，极端情况下，其他两个节点同时宕机了，只要有一个节点的数据最终写入磁盘了，那么其也是有一份完整的数据。

前面提到的 leader 向 follower 写入数据，这是文件的数据，而写入数据成功后，要向元数据中心（MetaServer）中写入文件的元数据信息，其会记录文件的大小，创建时间等信息，文件的元数据信息写入，往往具有强一致性，也就是必须保证有数据落盘，因为文件的元数据信息是非常关键的，这也是分布式与单机系统的差异之一。

最后有一个 quorum NRW 的数值之间的比较问题来分享一下。其中 N 表示副本数，通常是三副本，即 N=3；W 表示成功完成 W 个副本更新，才完成写入操作；R 表示读取一个数据对象时需要读 R 个副本。根据其数值的不同，有用一些数值关系来表示系统的一致性情况。

➢ 当 W + R > N 的时候，对于客户端来讲，整个系统能保证强一致性，一定能返回更新后的那份数据。

➢ 当 W + R ≤ N 的时候，对于客户端来讲，整个系统不能保证强一致性，可能会返回旧数据，但是也可以相对优化读写请求延迟。

3.2.3 两副本可靠吗

前面提到的分布式环境下的数据通常是三副本的，但是对于三副本来说，一份数据写入三个节点，有效利用率只有三分之一，对于一些相对不太重要的数据来说，三副本所带来的存储空间开销是比较大的，因此就会有一些新的思考，能否使用两副本来代替三副本？

对于两副本的出现，不管是数据的广播模型还是流式传递，

这里都是减少了一个数据节点，对于数据的流式传递来说，可能在极端情况下，会产生异常情况，下面来简单看看，如图 3–11 所示。

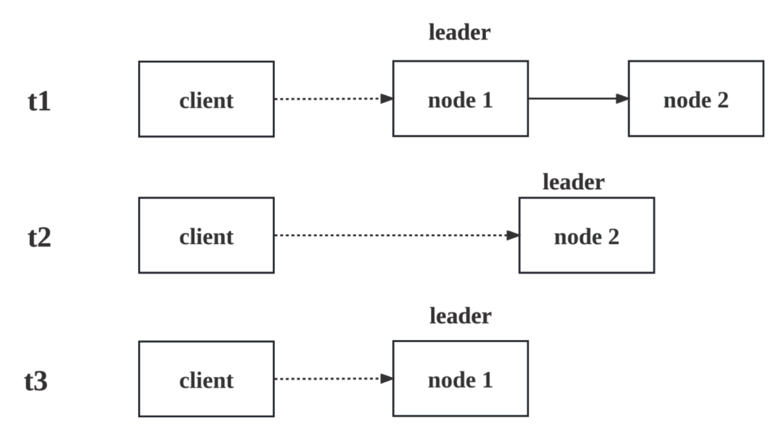


图 3–11 两副本模式数据写入图

➢ 在 t1 时刻，客户端向 leader 节点发送数据，然后准备传递到 follower 节点。

➢ 在 t2 时刻， node1 节点出现问题，导致数据无法传送过去，切换 leader 为 node2 并写入数据。

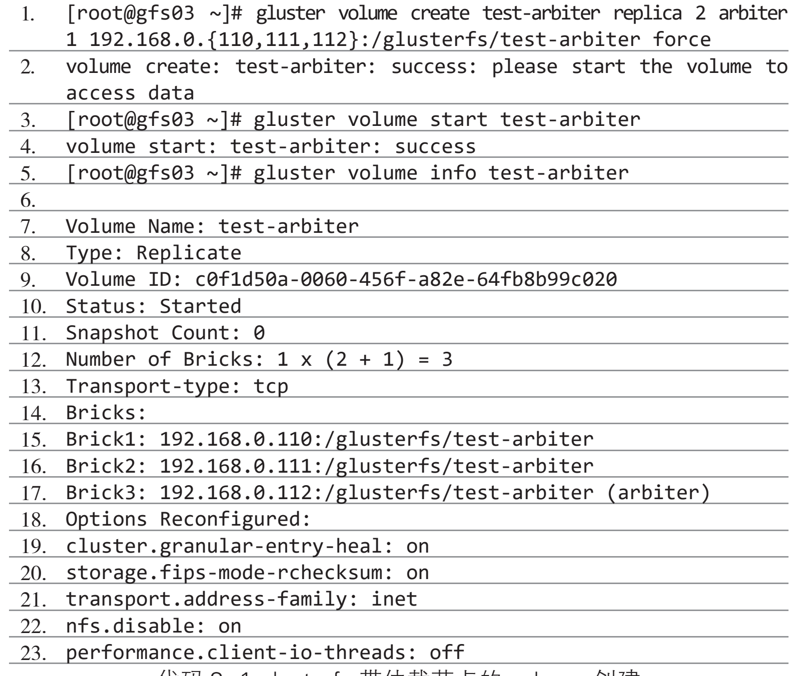
➢ 在 t3 时刻再次切换 leader 时，重新恢复为 node1，同时 node2 失联。

在这样的极端情况下，若短时间内频繁切换 leader，如果按照一份数据写入成功即可，那么这时候必然会导致数据之间出现互相覆盖的情况，并且可能没有一个节点会有完整的数据。因此为了解决这样的问题，一些系统需要保证两副本节点都写入成功才算完成。

同时因为两副本节点可能在其中一个节点出现宕机或者异常后，会使文件元数据信息不完整，因此在 glusterfs 中，为了解决该问题，增加文件元数据的冗余度，会额外增加一个节点叫

仲裁节点（arbiter），同时也可减少数据脑裂的风险。

下面来简单感受一下 glusterfs 中仲裁节点的使用（这里使用的版本是 glusterfs 9.x）。



```
1.  [root@gfs03 ~]# gluster volume create test-arbiter replica 2 arbiter
    1 192.168.0.{110,111,112}:/glusterfs/test-arbiter force
2.  volume create: test-arbiter: success: please start the volume to
    access data
3.  [root@gfs03 ~]# gluster volume start test-arbiter
4.  volume start: test-arbiter: success
5.  [root@gfs03 ~]# gluster volume info test-arbiter
6.
7.  Volume Name: test-arbiter
8.  Type: Replicate
9.  Volume ID: c0f1d50a-0060-456f-a82e-64fb8b99c020
10. Status: Started
11. Snapshot Count: 0
12. Number of Bricks: 1 x (2 + 1) = 3
13. Transport-type: tcp
14. Bricks:
15. Brick1: 192.168.0.110:/glusterfs/test-arbiter
16. Brick2: 192.168.0.111:/glusterfs/test-arbiter
17. Brick3: 192.168.0.112:/glusterfs/test-arbiter (arbiter)
18. Options Reconfigured:
19. cluster.granular-entry-heal: on
20. storage.fips-mode-rchecksum: on
21. transport.address-family: inet
22. nfs.disable: on
23. performance.client-io-threads: off
```

代码 3–1 glusterfs 带仲裁节点的 volume 创建

可以看到在上述代码中，Brick3 节点带有一个 arbiter 的标志，该标志就说明该节点是仲裁节点，该节点并不会保存数据，但是会保存与文件的元数据相关的信息。

为了更好地感受，下面进行简单的操作来查看一下。

```
1.  [root@gfs03 ~]# mkdir -p /mnt/test-arbiter
2.  [root@gfs03 ~]# mount -t glusterfs 192.168.0.110:test-
    arbiter /mnt/test-arbiter
3.  [root@gfs03 ~]# dd if=/dev/zero of=/mnt/test-arbiter/1.
    txt bs=64k count=1000
4.  1000+0 records in
5.  1000+0 records out
6.  65536000 bytes (66 MB) copied, 0.566256 s, 116 MB/s
7.
8.  [root@gfs03 ~]# du -sh /glusterfs/test-arbiter/
9.  16K     /glusterfs/test-arbiter/
10.
11. [root@gfs02 ~]# du -sh /glusterfs/test-arbiter/
12. 63M     /glusterfs/test-arbiter/
13.
14. [root@gfs01 ~]# du -sh /glusterfs/test-arbiter/
15. 63M     /glusterfs/test-arbiter/
```

代码 3.2

最后需要留意一下，glusterfs 中对仲裁节点的类型做了两种区别，分别是客户端模式和服务端模式，这里会涉及一些具体的参数差异，建议感兴趣的朋友可以自行查阅官网来详细了解和测试一下。

3.3 数据和请求负载均衡

了解了数据一致性问题后，下面就要考虑另外一个问题，关于分布式系统下的数据和请求负载均衡问题。这个问题对于单机系统来说通常遇到的比较少，而分布式下则需要有更多的考虑，而在讲解数据均衡问题之前，需要先来了解一下数据分组的 leader 集中所带来的问题。

3.3.1 leader 分散

这里要思考的问题是，如果一个机器下面有大量的磁盘，那么这些不同的磁盘可能属于不同的数据分组，其就可能有多

个数据分组的 leader 都是在相同的节点下，如图 3–12 所示。

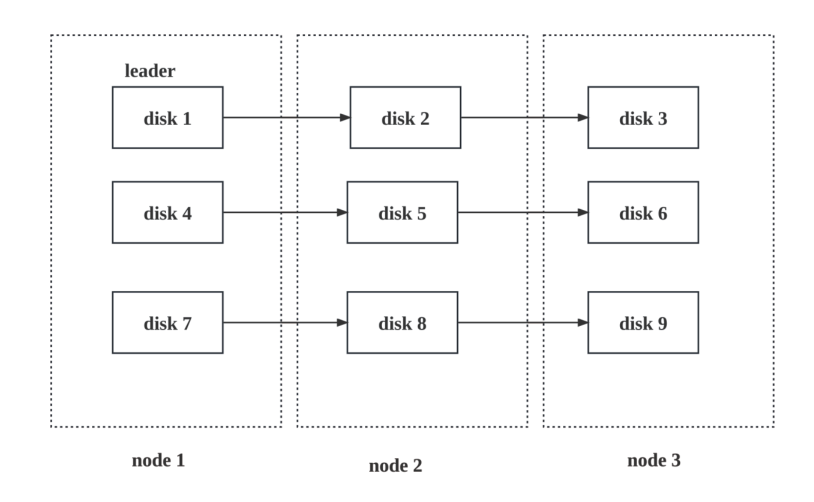


图 3–12

从前面的内容可以了解，leader 节点需要分发数据请求到 follower 节点中，同时还要在请求成功或者失败后，返回客户端请求。相对来说，leader 节点的性能压力会比其他节点要高，而一旦大量数据分组的 leader 集中在一个节点上，可能会导致该节点的系统负载较高，容易出现系统宕机或者卡死等现象，为了解决这个问题，需要对 leader 节点进行打散处理，通常的做法是对其进行随机或者轮训分配等，如图 3–13 所示。

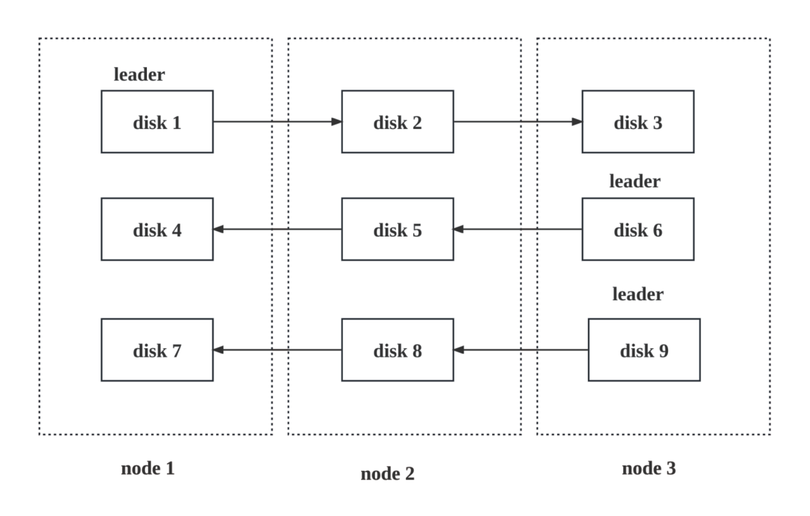


图 3-13

在系统经过长时间的运行后，会出现部分节点的宕机，leader 节点切换，其又会出现大量 leader 集中在某个节点上，这时候通常需要手动设置重新调整数据分组的 leader 节点，当然对于这种情况，在生产环境下建议在较少的数据流量下进行操作，因为一次 leader 切换的时间成本是比较高的，需要向配置中心告知，同时为了误判，有些系统还会使用监控中心来进行二次确认等。另外手动进行 leader 切换也需要有相应的命令支持的。

3.3.2 数据扩容和重平衡

对于一个存储系统来说，当每个分组的数据容量使用到一定程度时，都不可避免地要考虑如何增加新的存储空间。对于增加新的存储空间通常有两种办法，一是选择数据扩容，二是选择直接增加一组新的数据分组。对于这两种方式都有相应的优缺点，下面来探讨一下数据的扩容问题。

对于数据的扩容问题，在单机文件系统中，通常是添加一个或多个磁盘进入一个存储池里，而在分布式的三副本中，往往添加的磁盘个数，要和副本个数成倍数关系，这样难以出现数据倾斜，同时也减轻了数据写入时，需要考虑的磁盘权重比例等问题的复杂度。

而在 glusterfs 当中，如果 glusterfs 的 volume 是使用 heketi 创建的（一个管理工具，目前已经暂停维护，以前是 glusterfs 官方推荐的），那么就无法正常地使用 quota 进行容量限制了。底层使用 lvm2 的时候，如果要扩缩容，有两种办法，一是对底层的 vg 和 lv 进行处理，但是这种方法风险比较高，一旦操作不慎，会直接影响原来的数据，而且改变底层的 vg 和 lv 容量，信息还需要同步到上层的应用中，会比较麻烦。

二是使用 glusterfs 的 add-brick 功能，也就是通过增加数据节点的方式来进行扩缩容处理，下面来了解一下。

```
1.  root@gfs01:~# gluster volume info test-event
2.
3.  Volume Name: test-event
4.  Type: Distribute
5.  Volume ID: 4ff63ab9-561a-4c32-a4b9-60c5bda0c6e9
6.  Status: Created
7.  Snapshot Count: 0
8.  Number of Bricks: 1
9.  Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.9:/glusterfs/test-event
12. Options Reconfigured:
13. nfs.disable: on
14. transport.address-family: inet
15. storage.fips-mode-rchecksum: on
16.
17.
18. root@gfs01:~# gluster volume add-brick test-event 10.0.12.2:/
    glusterfs/test-event force
19. volume add-brick: success
20.
21. root@gfs01:~# gluster volume info test-event
22.
```

```
23. Volume Name: test-event
24. Type: Distribute
25. Volume ID: 4ff63ab9-561a-4c32-a4b9-60c5bda0c6e9
26. Status: Created
27. Snapshot Count: 0
28. Number of Bricks: 2
29. Transport-type: tcp
30. Bricks:
31. Brick1: 10.0.12.9:/glusterfs/test-event
32. Brick2: 10.0.12.2:/glusterfs/test-event
33. Options Reconfigured:
34. nfs.disable: on
35. transport.address-family: inet
36. storage.fips-mode-rchecksum: on
```

代码 3-3

这里有一个单 brick（brick 可以理解为对应的副本节点）的卷 test-event 使用了 add-brick 进行扩容。对于这样的扩容方式，不可避免地要考虑是否要进行数据重平衡（rebalance），因为可能之前旧的节点数据比较多，而新添加的节点数据很少。glusterfs 采用的是哈希的方式，可能会出现数据倾斜的问题。这一点是要非常注意的，是扩容后的一个潜在隐患问题。

为了解决这个问题，glusterfs 提供了手动进行重平衡的命令，如下所示。

```
1. root@gfs01:/mnt/rebalance-test# gluster volume rebalance
   rebalance-test start
2.
3. volume rebalance: rebalance-test: success: Rebalance on
4. rebalance-test has been started successfully. Use rebalance
   status command to check status of the rebalance process.
```

代码 3-4

当然对于很多知名的开源存储系统，如 glusterfs 和 ceph，都有对应的扩容命令，而扩容后若进行数据重平衡，则会延伸出一个问题，无法预估数据重平衡的时间。

某个系统扩容之后，数据重平衡时间很长，动辄以天甚至以周为单位，这对业务系统的影响很大，同时在数据重平衡过程中，对系统的性能影响也不小。

为什么会出现这样的问题呢？一个重要的原因是数据在重平衡的时候，需要重新计算数据的分布，同时可能会把大量的旧数据迁移到新的节点中，同时也要更新元数据中心的信息等。另外在数据重平衡时，业务也要能够保证正常读写，因此可能对于不同的系统，在重平衡时会做相应的数据保护，例如若有正在读写的数据，会暂停数据迁移，而已经正在迁移的数据，需要先保证数据迁移完成再进行修改，只能允许读取等。此外如果在数据重平衡过程中，出现了节点宕机的情况，那么必然会增加数据迁移的复杂度和系统处理难度，也会延迟数据重平衡的完成时间，若节点宕机时间过长，甚至会导致重平衡任务无法完成执行。

相对于数据节点的扩容，还有一种比较好的解决办法是增加数据分组，对于三副本为一组的数据分组来说，不会再对该分组进行扩容，只会新增加一个相同的三副本分组。对于这样的方式，需要上层业务来自行维护好数据写入的情况，或者通过客户端来对不同数据分组的磁盘容量权重进行排序等，优先把新数据写入新添加的数据分组中，这样的系统扩容方式，相较于前面，会减少很多运维上的麻烦。如果数据是有规律的，例如周期性的视频文件数据，可以提前规划好下一个时间周期，例如以每周为单位，当前周的数据写入当前的数据分组中，这样的方式会减少一些运维难度，但是需要提前评估好数据容量情况。

最后提醒一下，既然有数据的扩容，那么必然也会有对应的缩容，虽然缩容的情况相对少见，但是在 glusterfs 中也提供了相应的命令，也就是 remove-brick。一旦执行了数据的缩容，会

自动进行数据重平衡，这一点是需要留意的。

3.4 EC 卷

随着互联网的不断发展，大数据时代下，数据量越来越庞大，也让很多分布式存储系统对数据空间的焦虑问题越来越严重，同时也为了节省存储空间，达到节省成本的目的，急需一种高效的分布式卷来进一步节省存储空间，这时候 EC 卷就逐渐被大众所认识和了解。

3.4.1 背景及原理

在分享 EC 卷之前，想先分享一下分布式条带卷。顾名思义，所谓的条带卷，就是简单地把数据拆分成均等的多份数据分片（stripe），从而达到节省空间并利用多个分片节点 IO 读写并发的能力，但是对于分布式条带卷，其中带来的一个问题是无法保证数据的冗余和安全性，也就是说当部分节点出现宕机后，可能会使数据无法挽回，这是存储系统无法容忍的底线。而 glusterfs 在以前的历史旧版本中，也曾经使用和出现过条带卷。目前 lustre 系统中，也有该应用。

EC 卷和分布式条带卷的不同之处在于 EC 卷利用的是纠删码原理（erasure coding，EC），在对数据进行拆分之前，需要对数据进行编码，计算出校验码，分别存储到不同节点上，若其中节点出现宕机时，还可以保证数据的完整性。

纠删码通常使用的原理是 Reed-Solomon（RS）码，这是存储系统较为常用的一种纠删码，它有 k 和 m 两个参数，记为 RS（k，m）。k 个数据块组成一个向量 D 被乘上一个生成矩阵 B 从而得到一个数据向量，该向量由 k 个数据块和 m 个校验块构成。如果一个数据块丢失，可以通过一系列计算来恢复出丢失的数据块。RS（k，m）最多可容忍 m 个的块（包括数据块和校验块）丢失。

RS 编码的原理是如何求可逆矩阵的问题，如果把数据看成一个很大的矩阵（大家可以简单回忆一下线性代数），那么如何保证矩阵是可逆的，并且如何求解矩阵则是关键。对于前者，通常会把矩阵构建成一个范德蒙矩阵或者柯西矩阵，而后者则是使用伽罗瓦域来进行解决。本书并不是专门讲解 RS 编码原理的，因此如果想更进一步了解 RS 编码的数学原理，感兴趣的朋友可以在网上自行查阅相关资料或论文。

ceph 中的 EC 编码是以插件的形式来提供的，目前默认使用的是 Jerasure 库。除了 Jerasure，ceph 中还有 isa 和 lrc 等方式，而 isa 则是英特尔提供的 EC 库，只适用于其上的 cpu。下面来感受一下 ceph 中的 EC 命令参数。

```
1.  ceph osd erasure-code-profile set {name} \
2.          plugin=jerasure \
3.          k={data-chunks} \
4.          m={coding-chunks} \
5.          technique={reed_sol_van|reed_sol_r6_op|cauchy_orig|cauchy_good|
    liberation|blaum_roth|liber8tion} \
6.          [crush-root={root}] \
7.          [crush-failure-domain={bucket-type}] \
8.          [crush-device-class={device-class}] \
9.          [directory={directory}] \
10.         [--force]
```

代码 3-5

对于 HDFS，目前也提供了纠删码功能（hadoop 3.x 版本），其中常用的参数有 RS-3-2-1024K，RS-6-3-1024K 等，以 RS-6-3-1024K 为例，RS 代表使用了 RS 算法压缩，6 代表 6 个原始数据块，分别存在 6 个不同的 DataNode 节点中（数据节点）；3 代表 3 个校验块，也是存在不同的 DataNode 节点中，表示最多容忍 3 个节点的块丢失。同时还需要留意的是，如果丢失的是

校验块的数据，那么并不影响正常文件的读写，但是如果丢失的是原始数据块，那么需要读取校验块中的数据进行解压重新恢复数据。

3.4.2 EC 和多副本差异

对于 EC 卷来说，其优缺点也是非常明显的。其中优点是 EC 卷在写入大文件时，可以有效利用好文件分片多节点的 IO 并发能力，同时相比起三副本模式下，EC 卷若是 4+2（即 4 个数据分片，2 个校验码分片，最大允许 2 个分片数据丢失，也是最常见的方式之一），那么数据利用空间则仅比原始数据多出 2/（4+2）=1/3，同时还得到了一定的数据冗余可靠性。

对于 EC 卷来说，通常比较适合冷数据或者非重要数据。因为 EC 卷和多副本模式最大的不同之一在于，EC 中没有一个节点是拥有一份完整数据的，也就是说，每次读写数据的时候，都要先从所有的数据块节点读取分片数据，如果有失败的，则还要从校验块节点中读取信息，然后进行解码恢复。因此写入的时候，EC 卷中的文件数据的每一次写入，若是 4+2 的形式，则必须保证至少 4 个节点的数据写入成功才算最终成功。这对于读写请求的延迟来说，通常会比三副本节点更大。同时编码和解码的计算，还需要占用 cpu 资源。

除此之外，在代码架构实现层面，EC 卷是建立在分布式层面的，因此还需要增加 EC 文件抽象、分片管理、条带仲裁等功能。如分片管理，则是需要记录每个分片所在节点的相关信息；条带仲裁则是在数据异常修复时，对数据进行编解码恢复的。此外每个分片节点的数据还要写入本地文件系统中，而每个分片节点的数据则又是一个树形结构的数据，同时记录在本地文件系统中，并会有相关的 inode 信息，因此会有多层抽象的文件信息，如图 3–14 所示。

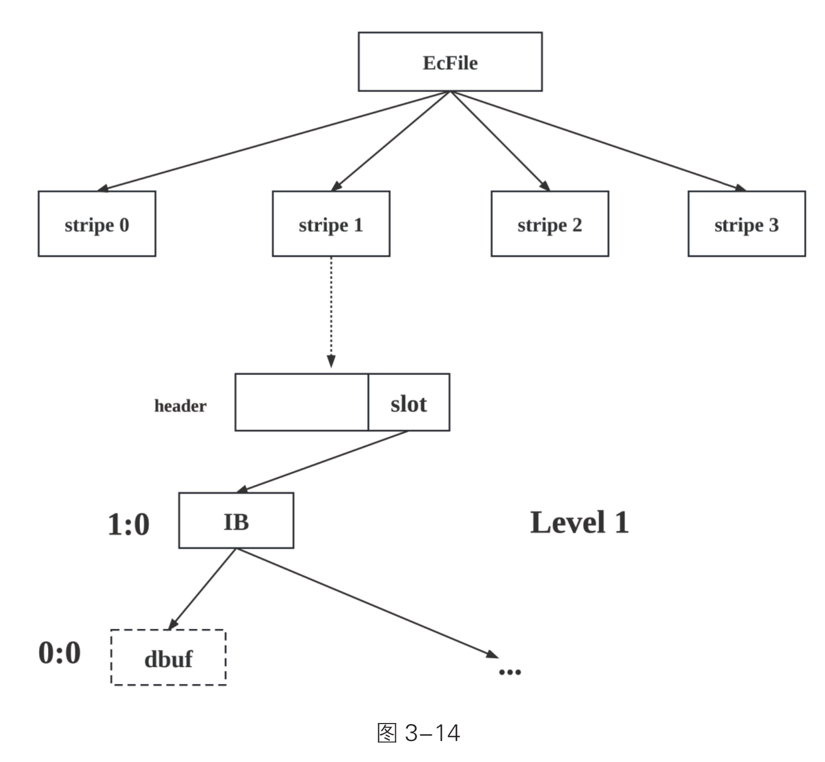


图 3–14

因此这里需要留意一个细节，每个分片节点上的数据文件大小，并不是真实的文件大小，往往只是其真实大小的 N 分之一。同时文件分片层与底层的数据映射应该保持一致，并且是一对一的形式。例如上层文件分片 id 为 0，表示文件第一个分片，则应有索引关系列表来记录对应文件系统中具体该分片的文件的 inode id。对于 EC 卷来说，会比副本模式多一层关系映射。

同时还需要留意，对于 EC 卷，因为每一个节点都只是拥有分片数据，并没有完整数据，所以每个节点的本地文件系统的快照功能，也并不适合用在 EC 卷上，很难使用本地文件系统的快照来恢复数据。

对于EC卷的使用，通常会用于冷数据或者归档文件数据上，因为这些数据的读写频率相对较少，同时对性能要求较低，但是需要节省存储空间。

最后还需要留意一下，使用了EC卷之后，如果系统要进行升级还是会有更多限制的。对于glusterfs来说，目前暂时并不支持使用EC卷的系统进行升级。因此要对EC卷的系统进行升级，通常为了安全，需要把数据迁移到新的集群中，对旧系统进行升级处理，但这样做必然会大量增加运维难度，同时也会导致系统升级时间过长和成本过高。

3.5 分布式异常数据修复

对于分布式系统来说，文件异常修复功能也是必不可少的，同时也非常考验系统的性能和稳定性，也是分布式文件系统中，一个比较难处理的模块。下面我们来探讨一下分布式系统中数据异常修复需要考虑和面对怎样的问题。

第一个需要思考的问题是如何知道哪些文件有异常呢？或者说，正在修改的文件信息，如果失败了，在哪里记录相关信息呢？关于这个问题，需要区分无中心和有中心架构两种。在一些书籍和资料中，对这个过程叫作change log，也可以叫作文件事务日志记录。

对于有中心的架构，我们先来重温一下正常的写入流程，如图3–15所示。

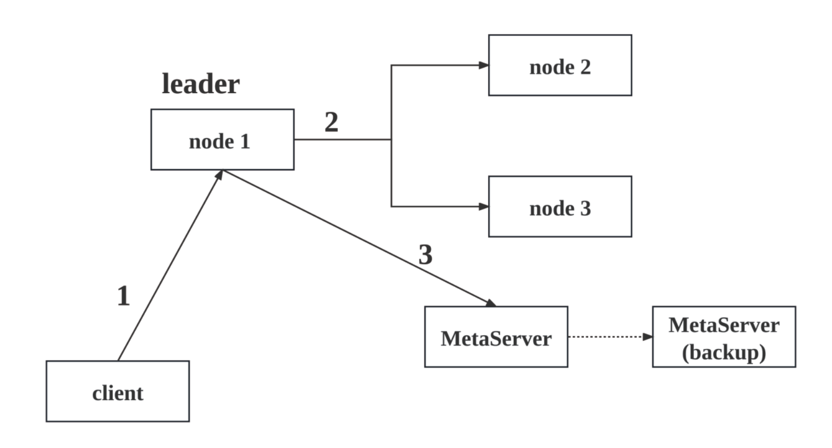


图3–15

（1）客户端会发送修改的请求，如write，setAttr，truncate等到leader节点中。

（2）leader本地会修改文件缓存，并且会发送请求到follower节点，即node2和3。

（3）三副本当中任意两个修改成功之后，由leader发送修改的文件元数据信息到元数据节点（MetaServer）。

若步骤（1）和（2）失败的话，请求是执行失败的。如果是其中某个节点执行失败了，但是符合quorum的话，那么leader节点也会告诉元数据中心，其中某个节点执行失败了，会有一条日志记录。

这是一种比较常见的架构执行方式，但是对于无中心架构来说，这里就缺少了第三步的模块，也就是元数据节点了，因此这里就需要寻找另外一种方式来替代这个过程，在glusterfs中则被称为AFR。具体的执行情况如下所示。

（1）锁定阶段：客户端在要执行的文件上，先获取一段文件的锁，防止其他客户端的同时写入出现并发冲突。

（2）预操作阶段：先在文件的扩展属性（trusted.afr.dirty）中

加 1，作为即将要执行的文件操作记录。

（3）操作阶段：数据节点（brick）对文件进行执行（FOP）。

（4）完成阶段：操作执行完成之后，对第二步的操作进行反向，也就是对扩展属性减 1。正常情况下，执行成功后，扩展属性的数值应该为 0。

（5）解锁阶段：释放文件锁。

从上述步骤可以看到，glusterfs 中其实也是利用了事务日志的概念来进行的，但是相对于常见的分布式系统来说，glusterfs 则是利用了文件的扩展属性来充当事务操作日志。一方面这样可以减少维护日志的空间和操作成本，同时也不用担心日志若丢失或者异常了，无法获取到操作过程。

当然对于 glusterfs 来说，上述的过程并没有那么粗糙，里面还有更多的细节。例如文件的锁可以是一个区间锁，那样可以做到部分情况下文件的并行修改。同时文件的扩展属性中，glusterfs 区分了对文件的元数据操作和数据操作部分，也就是说，可以通过扩展属性来更加精确地知道文件的异常部分是元数据信息还是文件数据信息。

下面通过一个简单的小例子来感受一下。

```
1.  root@gfs01:/mnt/test-afr# gluster volume info test-afr
2.  
3.  Volume Name: test-afr
4.  Type: Replicate
5.  Volume ID: 6637cea5-2ead-4c3f-a319-9ac7e32d3fd1
6.  Status: Started
7.  Snapshot Count: 0
8.  Number of Bricks: 1 x 3 = 3
9.  Transport-type: tcp
10. Bricks:
11. Brick1: 10.0.12.2:/glusterfs/test-afr
12. Brick2: 10.0.12.9:/glusterfs/test-afr
13. Brick3: 10.0.12.12:/glusterfs/test-afr
```



```
14. Options Reconfigured:
15. cluster.granular-entry-heal: on
16. storage.fips-mode-rchecksum: on
17. transport.address-family: inet
18. nfs.disable: on
19. performance.client-io-threads: off
```

代码 3-6

先创建一个 volume 然后挂载，接着再创建一个文件 1.txt，接着 kill 掉其中一个 brick 进程，再对 1.txt 文件使用进行修改，这里使用命令 date > 1.txt 导入时间，这时候对比一下正常和非正常的 brick 下的 1.txt 文件属性。

```
1.  //下面是正常brick
2.  root@gfs02:~# getfattr -d -m . -e hex /glusterfs/test-afr/1.txt
3.  getfattr: Removing leading ‘/’ from absolute path names
4.  # file: glusterfs/test-afr/1.txt
5.  trusted.afr.dirty=0x000000000000000000000000
6.  trusted.afr.test-afr-client-1=0x000000020000000000000000
7.  trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
8.  trusted.gfid2path.46104af11fea85fe=0x30303030303030302d303030
    302d303030302d303030302d3030303030303030303030312f312e747874
9.  trusted.glusterfs.mdata=0x01000000000000000000000000060e540820
    00000001b967e1d0000000060e54082000000001b967e1d0000000060e540
    5b000000003b3d42a9
10. 
11. root@gfs01:/mnt/test-afr# getfattr -d -m . -e hex /glusterfs/test-afr/1.txt
12. getfattr: Removing leading ‘/’ from absolute path names
13. # file: glusterfs/test-afr/1.txt
14. trusted.afr.dirty=0x000000000000000000000000
15. trusted.afr.test-afr-client-1=0x000000020000000000000000
16. trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
17. trusted.gfid2path.46104af11fea85fe=0x30303030303030302d303030
    302d303030302d303030302d3030303030303030303030312f312e747874
18. trusted.glusterfs.mdata=0x01000000000000000000000000060e540820
    00000001b967e1d0000000060e54082000000001b967e1d0000000060e540
    5b000000003b3d42a9
19. 
20. //这里是异常的brick
21. root@gfs03:~# getfattr -d -m . -e hex /glusterfs/test-afr/1.txt
22. getfattr: Removing leading ‘/’ from absolute path names
```

```
23. # file: glusterfs/test-afr/1.txt
24. trusted.gfid=0x21ea642138cc475e850a8bc7e1c5cc5c
25. trusted.gfid2path.46104af11fea85fe=0x30303030303030302d303030
    302d303030302d303030302d30303030303030303030312f312e747874
26. trusted.glusterfs.mdata=0x010000000000000000000000060e5405b0
    00000003b3d42a90000000060e5405b000000003b3d42a90000000060e540
    5b000000003b3d42a9
```

代码 3–7

从这里可以看到，首先异常的 brick 这里没有显示扩展属性 trusted.afr.dirty，接着正常的 brick 的文件扩展属性，这里的扩展属性为 trusted.afr.test–afr–client–1，其表示该 volume 名为 test–afr 的第二个 brick 出现异常了，接着可以查看一下 heal info 信息，对其进行核实。

```
1.  root@gfs02:~# gluster volume heal test-afr info
2.  Brick 10.0.12.2:/glusterfs/test-afr
3.  /1.txt
4.  Status: Connected
5.  Number of entries: 1
6.  
7.  Brick 10.0.12.9:/glusterfs/test-afr
8.  Status: Transport endpoint is not connected
9.  Number of entries: -
10. 
11. Brick 10.0.12.12:/glusterfs/test-afr
12. /1.txt
13. Status: Connected
14. Number of entries: 1
```

代码 3–8

除了上述的内容，接着我们需要思考第二个问题，当文件正处于修复的过程中，能否进行读写？这个问题需要根据不同的系统情况来判断，最简单的方式就是当文件正处于修复过程中，并不提供读写访问，只等待修复完成后才进行。这种方式的好处就是处理逻辑简单，但是可能文件修复的过程会因为文件很大，例如上百吉字节，那么掉线时间很长的节点，文件修

复时间可能会非常久，对业务系统影响很大。因此这种方式往往也比较少采用。

还有一些常见的处理方式是对数据进行分片，如 HDFS 中对文件以 MB 级别为单位大小进行分片，当文件数据异常时，非异常的数据分片还是可以提供读写访问的，这样可以进一步提高文件读写并行度。

接着下面来看一个极端的例子，leader 切换导致的数据超时异常问题，如图 3–16 所示。

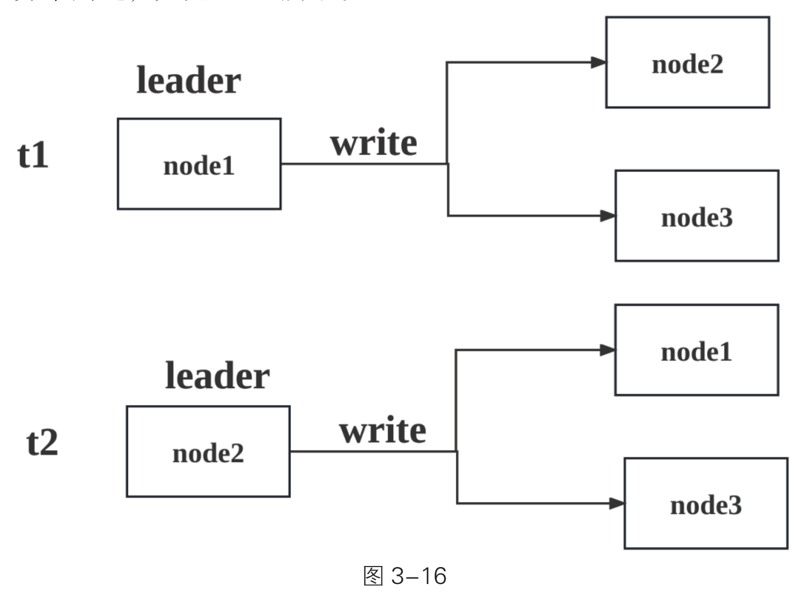


图 3–16

在 t1 时刻，其有一个正常的写入请求，以 leader 为 node1 节点执行，发送到 node2 和 node3 节点。在 t2 时刻发生了 leader 切换，恰好这时 node2 已经接受了该请求，新的系统拓扑结构发生了变化，因此 node2 节点可能还会把其继续下发到 node1 和 node3。以此类推，若发生极端情况，leader 频繁切换的时候，可能会使数据写入超时异常，无法执行完成。解决这个问题的方法有很多，例如在发送请求报文中加入已经完成的节点的信

息，在下发到下游时进行校验。

这是一个比较简单的例子，若在真实的复杂生产环境中，当 leader 切换时，可能有不同的任务会互相发送，例如旧的 leader 发送了异常数据删除请求，而新的 leader 发送数据写入请求，那么当互相发送数据请求时，可能会有一些无法预估的情况出现。因此对于 leader 切换时的情况，需要做大量的复杂场景测试，才能将问题暴露出来。

3.6 对象与文件存储差异

互联网的发展，数据爆炸式的增长，使文件存储面临着巨大的压力，因此需要拥有一种高扩展性、高可用和可靠的系统来应对海量数据存储，这时对象存储就出现了，也称为“面向对象的存储”。关于对象存储，大家经常听到的词语可能是扁平化的、高扩展性的、拥有卓越性能等。同时也有很多朋友会对文件存储和对象存储的差异感到迷茫，或者说并不太清楚到底对象存储和文件存储的差异在哪里。现在我们就来分享一下这二者的区别。

首先我们需要来了解一下，通常对象存储系统中会有哪些模块。常见的对象存储系统中，会有一些数据分组的功能，例如桶（Buket）、区域（zone）等概念，这些概念主要是为了集群中的数据划分和分组可以拥有更多和更加细致的划分度。同时有些是为了方便多区域部署集群，做好区域数据容灾等。

除此之外，对象存储中常见的还有权限管理功能，相比于文件系统的权限，在 Linux 文件系统中，使用 usergroup 的概念划分会显得相对粗糙，而对象存储中，有些系统会有更加细致的权限划分，例如匿名用户和授权用户等概念的使用。

同时在对象存储系统中，还会有文件生命周期管理，索引

子系统、存储子系统等模块功能，其中一些还会基于文件生命周期来为冷热数据存储和归档文件等功能。而数据索引生命周期管理功能，在 elasticsearch 等系统中会有使用到，其中分为以下 4 个阶段。

Elasticsearch 则定义了索引生命周期的 5 个阶段，其内容如下。

- Hot（热）：索引处于活动状态，能够更新（增改删）和查询。
- Warm（暖）：处于该阶段的索引不再支持更新，但是能够被查询。
- Cold（冷）：该阶段的索引不再支持更新，只能支持很少的查询，查询较慢。
- Frozen（冻结）：该阶段的索引不再支持更新，也很少查询，查询很慢。
- Delete（删除）：索引不再需要可以被安全删除。

存储系统的冷热数据分离功能的出现，可以基于不同的硬件设备性能来构建不同的存储系统，例如在热数据存储系统上，可使用高速存储设备如 ssd，固态的设备容量较小，但是性能较快。而在冷数据存储系统上，则可使用机械硬盘进行存储，同时还可以考虑使用 EC 卷来节省存储空间等。

除了这些内容，对象存储系统和文件系统在语义层面也有差异，其中一个较大差异是对象存储系统往往不支持文件对象的 truncate 功能，因为 truncate 功能会使文件分片删除的数据是非对齐的，例如文件系统的数据块是以 4k 为单位的，若一个 16k 的文件则有 4*4k 个数据块，若使用 truncate 功能，把文件缩小到 15k，则必然会无法和数据块对齐，需要做额外的数据填

充操作等。同时一般对象存储也不支持对象文件的跳跃写，因为当文件 truncate 到很大之后，若使用跳跃写，指定偏移 offset 写入时，可能会造成文件空洞，这往往会影响到文件性能和存储效率等。

目前主流的对象存储系统，如 Minio 还对一些功能特性做了约束，例如集群在线扩容功能，只能新增集群来增加容量。因为从前面的内容可以知道，数据的扩容，往往伴随着数据重平衡的使用，这二者给性能和场景处理带来了非常大的挑战。

4. 测试与优化

前面分享了很多关于文件系统的知识，从单机到分布式文件系统，接下来让我们转换视野，来了解一下文件系统中的测试与优化到底有哪些内容。

4.1 文件系统测试

4.1.1 单元测试

首先我们需要来了解一下什么是单元测试，单元测试是软件工程中降低开发成本，提高软件质量常用方式之一，单元测试是一项由开发人员或者测试人员对程序模块的正确性进行检验测试的工作，用于检查被测试代码的功能是否正确，养成单元测试的习惯，这不但可以提高代码的质量，还可以提升自己的编程和技巧。单元测试其实就是对模块、类、函数实现的功能执行检测，检测是否满足预期，是否达到功能要求，它是一次检查检验的过程。如果某个模块或者函数满足预期，则表示测试通过，否则表示失败，比如工厂在组装一台电视机之前，会对每个元件都进行测试看是否合格，这就是单元测试。

接下来我们需要来思考一下在文件系统中，需要对哪些模块进行单元测试呢？我们可以先来回顾一下 zfs 的整体架构，如图 4–1 所示。

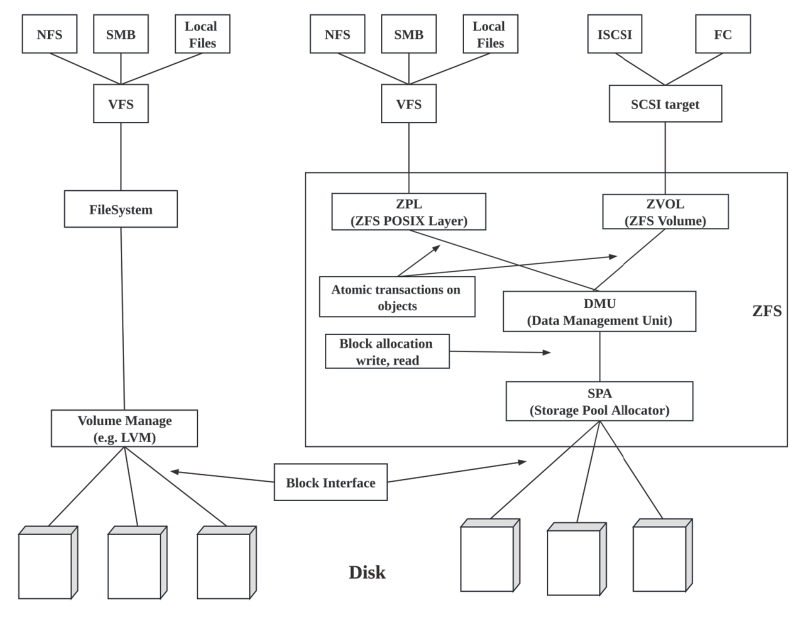


图 4–1

图 4–1 左侧是一个普通文件系统，右侧则是 zfs 的模块内容，其中包含 zpl、zvol、dmu、spa 等模块，除此之外还有事务组 txg，磁盘空间管理 metaslab 等，这些不同的模块构成了目前经典的 zfs 架构。

既然要做单元测试，必然需要考虑每个不同模块的功能完善与否，以正常读写来说，编写单元测试的时候，除了要做正常文件的读写测试，还要考虑文件跨越不同文件层级的变化，例如是否有 truncate，文件是从 level0 变为 level N（N 大于等于 1）。又或者反过来，当文件从 level N 被 truncate 到 level N–1 时的情况。除此之外，还要考虑文件的属性修改，如对扩展属性信息（attr）进行修改，其中 zfs 的 zap 模块还对扩展属性有长度范围的要求，超出一定范围会转换类型，这些都是要考虑的。

除了正常的情况，还要考虑一些异常情况下的请求是否正常，

例如空文件创建的时候，对系统进行重启，对日志进行重放，其能否保证文件不丢失等。若磁盘空间满了，修改请求无法再申请磁盘空间，是否应该返回符合预期的错误和处理结果。

对于单元测试来说，每次系统项目功能的修改，都要先保证能运行的单元测试用例是符合预期或者成功的，这样才能进入下一步的集成测试或者功能回归测试当中，也是研发用于自检功能完善的必备条件。

4.1.2 模拟真实环境测试

上面提到的单元测试，往往是对单一模块功能进行测试，这是一个非常难做的综合测试，例如写入大量文件，进行并发操作查看是否异常等，而通常部署一个分布式存储系统，往往需要部署多个不同的组件模块，因此需要有一种快速部署和测试的方式，docker 则是其中比较好的选择之一。

在使用 docker 进行部署的时候，为了模拟真实环境的操作，例如三副本模式下的数据交互情况，可以启动 3 个 docker 容器来作为不同的节点，再启动一个 docker 来作为元数据节点进行交互，这样就构成了一个基础简单的真实测试环境了，如图 4–2 所示。

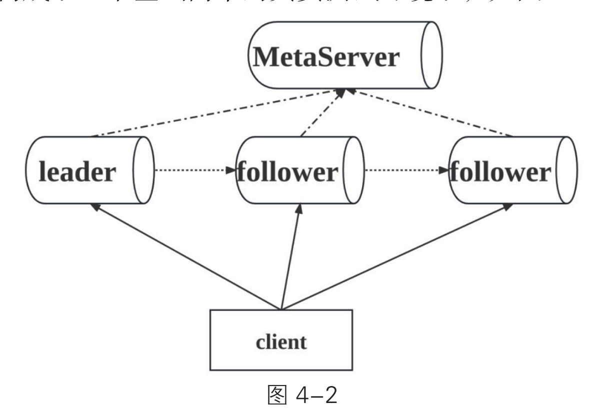


图 4–2

当然需要注意的是，docker 模拟真实环境的测试并不能完全替代真实物理机测试，尤其是磁盘的热插拔、网络隔离、节点宕机等情况。在真实环境下，当机械盘被拔出后，可能会使磁盘中的缓存数据丢失，但是在 docker 容器环境中则很难测试复现。

在经过 docker 的模拟测试之后，如果没有测试出 bug 和问题，那么通常会认为一个功能的修改，已经得到了基本的功能性测试覆盖保障，接下来就可以提交代码进行集成测试了。通常会使用 Jenkins 等工具来检测，等代码提交后，自动运行一些模拟故障的测试脚本来进行测试。在部分有条件的情况下，还会使用 nginx 来部署两套环境，一是真实的生产环境，二是测试环境，使用流量镜像功能，让部分流量复制到测试环境中，定时执行一些随机故障测试内容，或者使用 chaos Mesh 等混沌测试工具来模拟故障，达到充分测试的场景。

最后还需要留意一下，以上提到的测试内容，大部分都是软件代码层面的测试情况，而存储系统中，硬件故障也是一个必不可少的测试场景，但是相对于软件模拟故障测试，硬件测试则会比较难以模拟，如固态硬盘的加速老化、磁盘的坏块增加、磁盘读写性能下降等，虽然部分混沌测试工具可以通过添加请求随机休眠的做法来模拟，但是仍然无法替代真实的场景进行测试，而这些也是存储系统测试中比较难以测试覆盖的场景。

4.1.3 生态工具测试

除了以上提到的测试，生态工具相关的测试也是必不可少的，甚至很多存储系统的优劣，还有使用群体的多少，都会与生态工具有关，因为如果生态工具做得很差，那么其他团队也会抗拒使用该系统。而所谓的生态工具，其中包括如集成进 k8s

中的 CSI 插件、samba 插件和一些集群迁移等工具，这些工具与原系统相比往往是独立的，需要额外安装或者配置，同时这些工具的测试，如集群数据迁移工具，虽然平时使用很少，但是一旦使用了，就是不可逆的状态，往往很难停止下来，因此提前做好一些生态工具的测试也是必需的。

这里需要简单提醒一下，像 glusterfs 一些管理工具，在一段时间内，如果系统中停止使用 heketi 工具，但是又没有办法找到更完善优秀的工具来进行代替，这也是一个比较尴尬的时刻，这也是自研存储系统人们需要避免的，尽量在替代工具完善之前再停止维护旧的工具。

为什么 glusterfs 会放弃 heketi 这样的管理工具而开发新的工具呢？下面来简单分享一下。

在使用 k8s+glusterfs 的架构中，k8s 管理 glusterfs 的 volume 工具，可以使用 heketi 来进行管理，而 heketi 的原理，其实是对块设备进行格式化，做成 lvm2，弄成 vg，而有 pod 要申请 volume 的时候，则要从 vg 中创建 lv，格式化成 xfs 文件系统格式，接下来挂载到系统的目录上面使用，对于这种方式，优缺点是非常明显的。优点是使用这种方式创建的 volume，非常方便进行容量监控，因为底层的 vg 信息创建出来的 lv 已经限制了容量，在挂载之后，可以直接使用 exporter 进行检测。同时使用这种方式创建的 volume，可以很方便地使用快照功能。

当然这样做的缺点也是显而易见的，一旦 volume 容量不足，因为底层是 lvm2 的，heketi 的扩容是再次从 vg 中创建一个新的 lv 出来，并且再次格式化挂载，对原来的 volume 进行 add-brick 操作，对于 volume 来说则需进行数据的 rebalance 操作，而 rebalance 操作则是一个不确定性非常大的操作，因为对于容量很大的 volume，或者小文件非常多的 volume，数据的重平衡 rebalance 操作时间是没有办法准确预估的。在测试环境中对

nexus 进行扩容操作，其时间花费过多，当然这个时间可以通过设置一些参数来减少，但是这仍然不是一个在短时间内可以完成的操作。另外对于重平衡后的 volume，没有办法完全做到数据的平均分布，尤其是一些数据大小并不是一致的时候，会出现数据倾斜的风险。

将使用块设备格式化为 lvm2，若多个服务都是部署在当前的块设备上，那么对于节点的读写负载是比较高的，同样会影响到其他服务的使用，而 heketi 为了解决这个问题，有一个 tag 标签的功能，可以为不同的块设备在格式化之后打上 tag 标签，storageclass 可以指定对应的标签的块设备创建 volume，实际上就是一个分组的功能，因此这里也需要额外规划好块设备的分组问题。

除此之外，像 glusterfs 中还有一些其他的工具如 Nfs-ganesha。Nfs-ganesha 是 NFS v3、4.0、4.1 和 4.2 的用户模式文件服务器，考虑使用这个项目的原因，是出于对 volume 的安全管理，对于 glusterfs 集群来说，只要知道了 volume 和任意一个节点的 ip，就可以随意挂载并且使用了，只要防火墙允许，但对于生产环境来说，如果一个 volume 是一个团队使用的话，那么权限管理就比较混乱和麻烦了，而使用 ganesha 项目则可以隐藏 volume 信息。当然，ganesha 项目是基于 NFS 使用的，而 NFS 对于读写性能的损耗也比较大，因此，如果在生产环境下使用的话，建议多进行压测。

当然目前还有一些非常优秀的第三方工具，如 JuiceFS。JuiceFS 是一款面向云原生设计的高性能分布式文件系统，在 Apache 2.0 开源协议下发布。提供完备的 POSIX 兼容性，可将几乎所有对象存储接入本地作为海量本地磁盘使用，也可同时在跨平台、跨地区的不同主机上挂载读写。

4.1.4 性能测试

对于性能测试，相信很多人都会有接触过，一些常见的测试工具命令有 fio 和 vdbench，其中使用 fio 可以测试多文件的并发读写情况，如下所示。

```
1. //随机读
2. #fio -filename=/mnt/xxx   -direct=3  -ioengine=libaio  -bs=4
   k -size=5G -numjobs=10  -iodepth=16 -runtime=60   -thread -rw
   =randread
3.
4. //顺序读
5. #fio -filename=/mnt/xxx   -direct=3  -ioengine=libaio  -bs=4
   k -size=5G -numjobs=10  -iodepth=16 -runtime=60   -thread -rw
   =read
6.
7. //顺序写
8. #fio -filename=/mnt/xxx   -direct=3  -ioengine=libaio  -bs=4
   k -size=5G -numjobs=10  -iodepth=16 -runtime=60   -thread -rw
   =write
```

代码 4-1

对于性能测试其文件的大小读写压测要符合实际生产需求。笔者曾有多位朋友咨询一些与存储系统性能相关的问题，往往最直接的是问，哪个系统的性能比较好？但是这样的问题是没有答案的，因为不同的需求有不同的系统可以选择。

对系统进行一系列压测的时，最常见的还是以 4KB 为数据大小进行的。但是实际上这并不会和公司生产数据相匹配，例如视频文件等数据通常以 GB 为单位，因此可以使用相对大一点的 block size 来进行测试，如 128KB 或者 1MB 等。因此，生产环境中的数据与实际不适配，则性能测试报告往往也容易得出该系统性能不好的结论。另外需要留意的是使用 fio 进行压测的时候，若指定写入的 bs 过小或者与文件系统的文件 block 不匹配，例如通常小于 4KB 或指定 fio 的 bs 为 5k，往往会使其性能

很差，这是可能是因为触发了文件的非对齐写，这样的测试效果通常是不太理想的，需要留意。

做性能测试的时候，也不建议使用很激进的文件压测方式，例如使用 fio 测试时，先创建上百万甚至上千万个文件在一个目录层级下，这样的测试通常过于激进也很少见，一旦文件数量在单目录下过大，那么使用 ls 命令也可能会耗费很长的时间，可以通过业务拆分等方式来进行优化，简单来说，往往性能测试需要对系统进行参数配置优化，这样才能够得到一个相对比较好的性能测试报告。

做性能测试和一般的功能性验证测试不同点在于要先保证研发的功能验证通过后再进行，而且做性能测试时，会对系统的参数进行一些优化，通常有以下方面。

- 调整大一点系统的缓存容量，避免数据过于频繁持久化。
- 降低系统的日志级别，减少日志输出量。
- 提高周期性运行的线程或者子模块时间，例如数据监控上报、数据异常定时扫描等线程的时间周期，避免频繁上报节点数据。

如果想要模拟硬件设备的一些故障测试，如性能衰减，那么可以创建多个块设备，写入大量的数据，同时再对文件系统进行压测。在这样的场景下，因为最终写入的都是相同的硬件设备，因此会出现 IO 抢占和 IO 性能瓶颈的情况，当大量读写请求出现延迟时，可以进一步验证系统的处理是否完善、超时机制的设置是否合理等。

4.2 小文件优化

小文件的性能优化一直以来都是存储系统的重难点，也是一个非常有挑战性的话题。在了解小文件优化之前，我们需要明确一下，何为小文件？是指该文件大小小于一个阈值，例如当文件小于 4KB，小于 128KB 等情况就可以称为小文件吗？其实并没有如此简单，通常来说，小于 4KB 的文件可以称为小文件，但是对于存储系统来说，小文件的真正定义应该是小于其基本的数据块单元大小。

以 HDFS 这些分布式系统为例，在 HDFS 中数据分片大小往往是以 MB 为单位的，因此，如果有文件的大小是小于该大小的，那么对于 HDFS 来说，该文件就是“小文件”了。而在单机文件系统中，以 zfs 和 xfs 为例，小于其 block size 大小的文件也可以被称为小文件了。

为什么需要考虑小文件的优化呢？一个直接的原因就是小文件的数据很少，如果按照正常的文件结构来存储，会造成很大的空间浪费。因此在 xfs 中会有对应的 short format 结构等，就是会把文件数据填充进文件头部中，达到更加紧凑的数据结构空间排列，避免再次申请一个新的数据块来保存数据造成浪费。

从而对于小文件的优化，除了在文件结构上进行优化，还可以考虑把多个小文件合并成一个大文件的方式进行优化，这种优化方式更加适用于分布式存储系统。如果数据写入后，数据很少会进行修改，那么其就适合这样的优化场景。若这些小文件被频繁的修改，那么可能会使大文件的中间数据出现空间碎片化和空洞的问题，同时大文件的异常修复也是难题。

4.3 lustre 对 zfs 参数优化

目前很多优秀的分布式系统会对依赖底层单机文件系统有一些建议的参数优化，其中 lustre 的 OSD 的类型可以是 zfs，官方文档中有一项内容则是对 zfs 的参数的优化，[11] 下面来了解一些参数的优化内容，如表 4-1 所示。

表 4-1

参数	建议值
zfs_vdev_scheduler	deadline
zfs_arc_max	75% RAM
zfs_vdev_async_read_max_active	16
zfs_vdev_async_write_active_min_dirty_percent	20
zfs_vdev_async_write_min_active	5
zfs_vdev_sync_read_min_active	16
zfs_vdev_sync_read_max_active	16

从上述的参数名称中可以看出，大部分情况下，lustre 对 zfs 的参数优化在于缓存中脏数据的读写性能阈值修改，主要目的是希望通过调大缓存的脏数据比例和可用内存比例等来提高文件数据读写的性能。除了 lustre 有对 zfs 进行参数优化，glusterfs 也会有一些类似的参数优化文档，各位感兴趣的朋友可以自行查阅一下。

附 录 引 用

[1] https://lwn.net/Articles/360199/

[2] https://github.com/torvalds/linux

[3] https://btrfs.wiki.kernel.org/index.php/Btrees

[4] https://github.com/openzfs/zfs/commit/d683ddbb7272a179da3918cc4f922d92a2195ba2

[5] https://docs.oracle.com/cd/E26505_01/html/E37384/docinfo.html#scrolltoc

[6] https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/ZIO%20Scheduler.html

[7] https://docs.gluster.org/en/latest/Upgrade-Guide/generic-upgrade-procedure/

[8] https://docs.gluster.org/en/latest/Administrator-Guide/Gluster-On-ZFS/#finish-zfs-configuration

[9] http://www.mckusick.com/bookrefs/zfs_dedup.html

[10] https://github.com/openzfs/zfs/pull/14037

[11] https://wiki.lustre.org/ZFS_Tunables_for_Lustre_Object_Storage_Servers_（OSS）

[12] https://farseerfc.me/zfs-layered-architecture-design.html